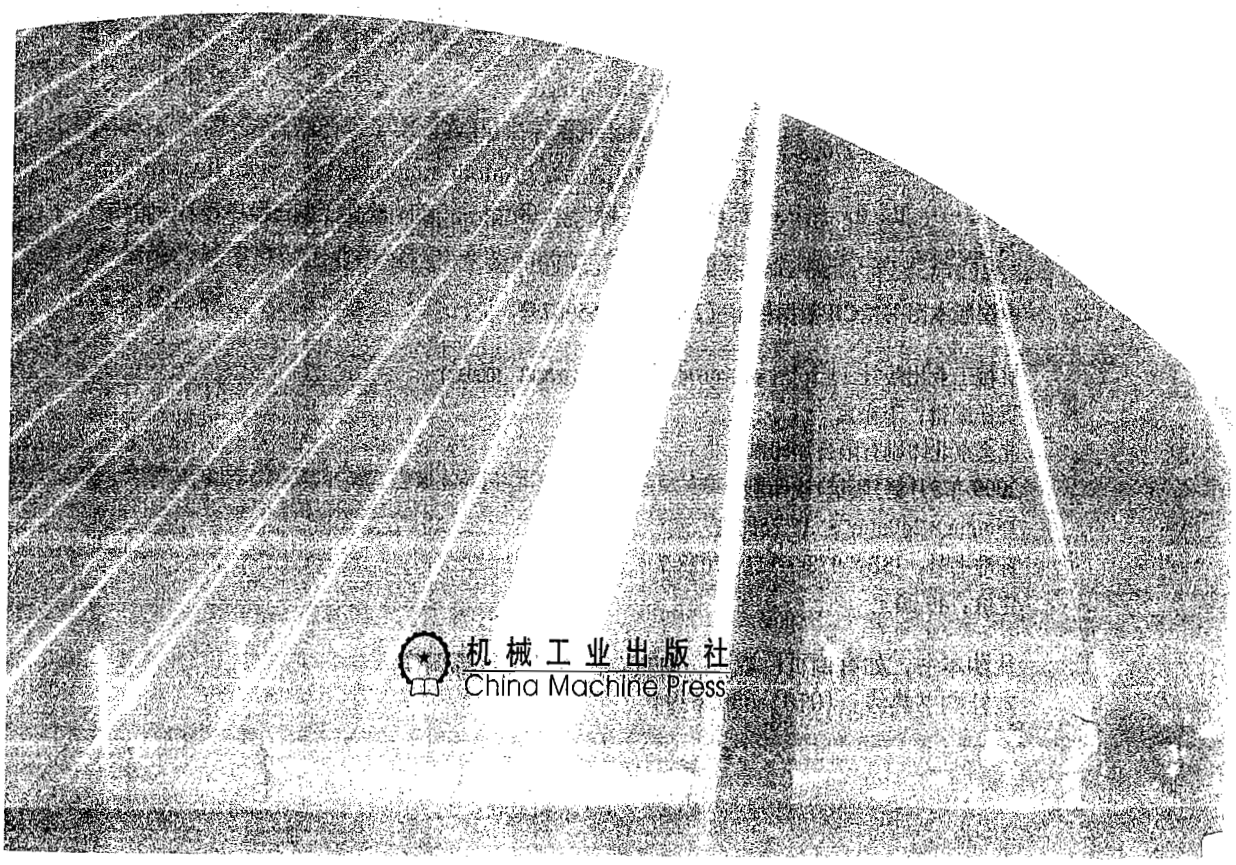
开发人员专业技术丛书

Python Web Development with Django

Django Web开发指南

Jeff Forcier
(美) Paul Bissex 著
Wesley Chun
徐旭铭 等译



本书讲述如何用Python框架Django构建出强大的Web解决方案，本书讲解了使用新的Django 1.0版的各种主要特性所需要的技术、工具以及概念。

全书分为12章和6个附录，内容包括，Django Python实战，Django速成：构建一个Blog，起始，定义和使用模型，URL、HTTP机制和视图，模板和表单处理，Photo Gallery，内容管理系统，Liveblog，Pastebin，高级Django编程，高级Django部署。附录内容包括命令行基础，安装运行Django，实用Django开发工具，发现、评估、使用Django应用程序，在Google App Engine上使用Django，参与Django项目。

本书适用于Python框架Django初学者，Django Web开发技术人员。

Simplified Chinese edition copyright © 2009 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Python Web Development with Django (ISBN 978-0-13-235613-8)* by Jeff Forcier,Paul Bissex,Wesley Chun, Copyright © 2009 .

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison-Wesley.

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2009-2370

图书在版编目（CIP）数据

Django Web开发指南 /（美）杰佛（Jeff, F.），（美）鲍尔（Paul, B.），（美）陈仲才（Wesley, C.）著；徐旭铭等译. —北京：机械工业出版社，2009.5

（开发人员专业技术丛书）

书名原文：Python Web Development with Django

ISBN 978-7-111-27028-7

Ⅰ. D…　Ⅱ. ① 杰…　② 鲍…　③ 陈…　④ 徐…　Ⅲ. 软件工具-程序设计-指南　Ⅳ. TP311.56-62

中国版本图书馆CIP数据核字（2009）第065987号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：李东震

北京京北印刷有限公司印刷

2009年5月第1版第1次印刷

186mm × 240mm · 18.25印张

标准书号：ISBN 978-7-111-27028-7

定价：49.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

译 者 序

计算机行业真是一个很有意思的行业，它每天都发明无数新名词却又拒绝它们进入生产领域。一项技术往往需要十数年的成长才会被业界慢慢接受，而其中的大部分还来不及成浪就已死在沙滩上了。Java当初熬了快十年才火起来，C++即使借了C的光也是多年媳妇熬成婆。Python比Java发明的更早，早期和Perl抢饭吃，虽有Zope/Plone这样出色的作品，依旧难逃曲高和寡的命运。

随着Web 2.0的改革号角吹响，开发人员们开始意识到，轻型的框架才是可持续发展的硬道理。一时间，小到只有一个文件的web.py，大到像Quixote等能支持豆瓣（douban.com）这样大型应用的平台百花齐放，连Python的创始人Guido van Rossum都忍不住跑来凑热闹，扬言要挑一个顺手的用用。那么最后到底是谁入了Guido的法眼呢？那就要隆重推出本书的主角——Django了！能被Guido看中那就算一只脚伸进了Google，这不，Google的新概念云计算的产品之一——Google App Engine（GAE）已率先支持了一个剪裁过的Django框架，本书会向你介绍如何在GAE上运行Django的程序。

相比其他Python Web框架，Django更为一体化，它安装简单且灵活多变，这很符合Python开箱即用的特点。选择Django，你无需安装其他组件就能写程序运行了，同时它的灵活性体现在每个部分都能拆下来换装其他组件。简而言之，它的集成度高又保持了松耦合，这一点是相当了不起的。开发人员可以在掌握了一些基础后，把各个组件换成自己顺手的工具，这对快速开发要求很高的Web 2.0来说至关重要。如果你还在Web开发的门口徘徊不决的话，不妨来试试Django，或者你可以访问CPUG的wiki来看看Python主流框架从而加以比较。

http://wiki.woodpecker.org.cn/moin/PyWebAppFrameworks

http://wiki.woodpecker.org.cn/moin/PyWebFrameVs

最后，感谢机械工业出版社华章公司的陈冀康老师和本书的编辑，没有你们就没有这本书中文版的出版。

徐旭铭

前　言

欢迎使用Django

欢迎来到Django的世界，很高兴能和你一起进行这趟旅程。你会发现有了这个强大的Web框架，做每件事情都变得便捷起来——从设计开发新应用到不用大刀阔斧地修改代码就能为现有代码提供新的特性和功能。

关于本书

市面上已经有了一些讲解Django的书籍，但是本书的特别之处在于它着重介绍的三个方面：Django基础、各种示例以及Django的进阶内容。我们希望写出一本关于这个主题最完整的教程，无论你的背景是什么都能读懂它。同时，你还能完整地了解这个框架以及它的能力。

章节指引

图P.1根据你对Python和Django的了解程度给出了不同的起点。当然，我们的建议是最好重头读到尾，不过如果时间比较紧张的话，这张图或许能提供给你一些帮助。不管你的水平如何，你都可以去阅读代码，毕竟这是学习了解应用程序最好的方法之一。此外我们还提供了一份详细的章节指引来帮助你决定从哪里开始阅读。

第一部分：入门

第一部分向Django以及Python的新用户介绍了基本的内容，不过即使是有经验的读者，我们也推荐您读一下第3章，“起始”。

第1章，Django Python实战

这一章为那些不了解Python的读者做了一下介绍。本章不仅展示了基本语法，还进一步深入介绍了Python的内存模型、数据类型，特别是在Django里大量使用的结构。

第2章，Django速成：构建一个Blog

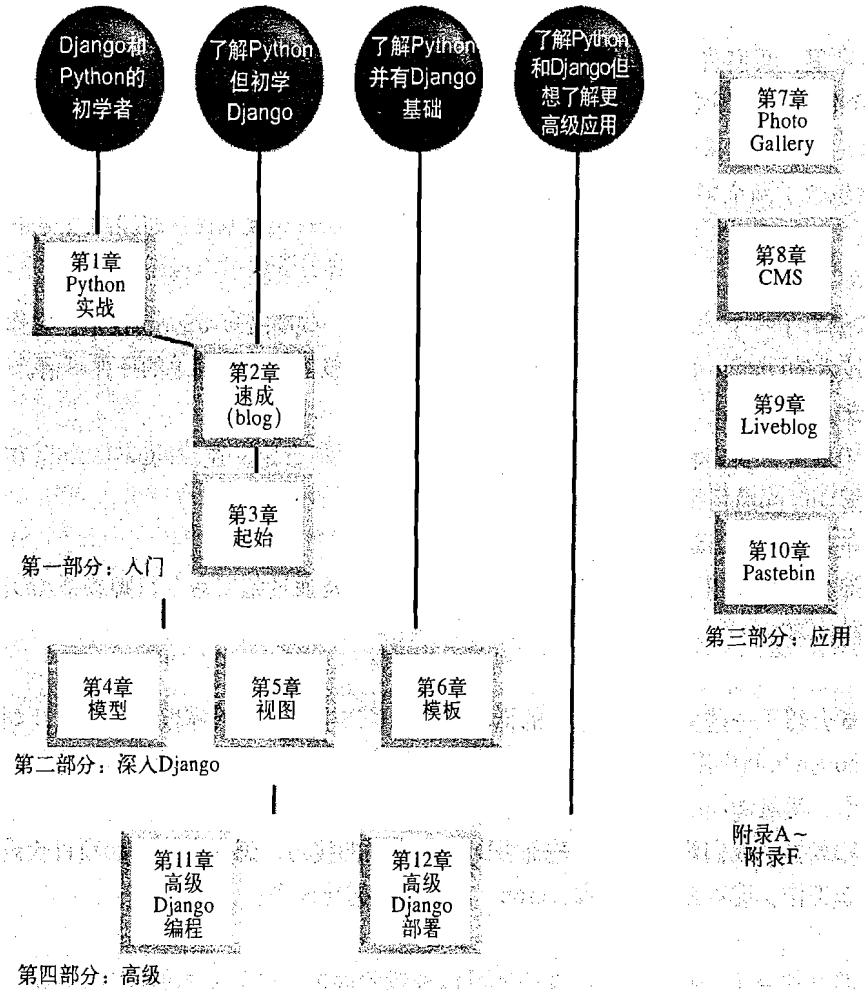
这一章是为那些希望跳过Python介绍，直接在15～20分钟完成一个Django应用的读者准备的。本章完美地展示了这个框架的强大能力。

第3章，起始

对那些更没耐性的读者，这一章介绍了开发Web应用基础的方方面面（对新手和老鸟都有益）。之后，我们会讲解这些概念是怎样和Django结合起来的、Django的设计哲学以及它独立于其他Web应用框架的魅力所在。

第二部分：深入Django

第二部分覆盖了框架所有基本组成部分。为第三部分中的例子打下了基础。



图P.1　根据你的Python以及Django经验所建议的阅读指引

第4章，定义和使用模型

在第4章里，我们将学习如何定义和使用数据模型（model），以及Django里对象关系映射（ORM）的基础，包括从简单的值到复杂的关系。

第5章，URL、HTTP机制和视图

这一章详细介绍了Django处理URL和HTTP协议的方法，这包括中间件层，以及如何使用Django简便快捷的通用视图（view）等。另外还介绍了怎样自定义或者重新写一个视图。

第6章，模板和表单处理

第6章介绍了框架最后一个主要的部分，Django的模板语言和它处理表单的机制。这包含了向用户显示数据以及从用户那里获取数据。

第三部分：Django应用实例

在第三部分里，我们将创建4个不同的应用，每一个都展示了Django开发中不同的部分和组件。它们将介绍一些新构想，并且扩展在第一部分和第二部分里讲解的概念。

第7章，Photo Gallery

在第7章里，我们将学习如何在你的URL结构上应用“不要重复自己”（DRY）的原则，以及创建一个包含图片预览的简单的照片浏览应用程序。

第8章，内容管理系统

第8章包含了两个创建CMS类系统的方法，还介绍了一些用户贡献的第三方Django应用的使用方法。

第9章，Liveblog

第9章编写了一个“liveblog”——这个站点使用了一些高级JavaScript技术。它作为一个应用了Ajax技术的Django项目，展示了使用任何Ajax工具包都是非常简单的一件事情。

第10章，Pastebin

在第10章里，我们将通过学习创建一个几乎没有任何自定义逻辑的pastebin（在线着色工具）来了解Django通用视图的力量。

第四部分：高级Django技术和特性

第四部分主要介绍了自定义Django的admin应用以及通过编写命令行脚本来和Django应用交互等高阶内容。

第11章，高级Django编程

第11章介绍了一些关于改进代码的话题，比如RSS生成、扩展模板语言，以及如何更好地使用Django的admin应用。

第12章，高级Django部署

在第12章里，我们将学习到一些部署Django应用的技巧，怎样在Django项目代码之外和应用程序一起工作，比如命令行脚本、cron、测试和数据导入等。

附录

附录部分集合了一些和本书有关但不足以成章的内容。学习基本的UNIX命令、Django安装和部署策略、开发工具等。

附录A，命令行基础

附录A介绍了基本的UNIX命令行。如果你从来没接触过的话，这个附录将对你非常有用。

附录B，安装运行Django

在附录B里，我们将学习安装所有运行Django所需的必要组件，包括各种可能的数据库和Web服务器，以及一些特定的部署策略的小技巧。

附录C，实用Django开发工具

附录C简要介绍了一些你可能熟悉也可能不熟悉的开发工具，比如代码控制、编辑器等。

附录D，发现、评估、使用Django应用程序

出色的程序员写代码，但是伟大的程序员重用代码！在附录D中，我们来分享一些关于到哪里以及如何寻找可重用的Django应用的经验。

附录E，在Google App Engine上使用Django

附录E从一个独特的视角出发，考察了Google新的App Engine如何使用Django，以及学习

如何把你的Django应用放到App Engine框架上去运行。

附录F，参与Django项目

附录F讨论了如何向Django项目贡献代码并且成为社区的一份子。

编写体例

在本书中，我们使用粗体来介绍新的和重要的术语，用斜体来表示强调，http://links/表示URL，用等宽字符来描述Python变量名和命令行命令等。如果有多行的代码和命令，则把它们包含在等宽字符的块里，像这样：

```
>>> print "This is Python!"
This is Python!
```

在撰写这本书和示例应用期间，我们用到了所有三个主要的平台——Mac OS X、Linux和Windows。另外，我们还测试了所有主要的浏览器（虽然可能没有在书里的图中显示出来），包括Firefox、Safari、Opera和Internet Explorer。

本书资源

你可以通过authors@withdjango.com来和作者联系。我们的网站http://withdjango.com 包含了大量辅助的资料，本书也在很多地方有所提及。

致　谢

虽然我的名字可能是排在作者列表里的头一个，但是如果没有另外两位作者的努力和贡献的话，这本书是不可能完成的。Paul和Wesley是两位出色的绅士学者，能和他们共事感觉很棒。

说到绅士学者，Django的核心团队也一样担得起这个名号。最开始的四大金刚——Adrian Holovaty、Jacob Kaplan-Moss、Simon Willison和Wilson Miner——为Django奠定了（而且继续在贡献）一份坚实的基础，在这之上，Malcolm Tredinnick、Georg Bauer、Luke Plant、Russell Keith-Magee和Robert Wittams进一步扩展了这个基础。要知道我不是一个容易被感动的人，但是这里的每一位都给了我很大的灵感。

我还要感谢另外两位“Django人”和IRC高手，Kevin Menard和James Bennett，以及NYCDjango组的各位朋友，这是Django社区里典型的牛人汇聚的地方。

最后，大力感谢Pearson的各位同仁，包括编辑和技术评论等（Wesley会提到你的！），这里特别要感谢排版的工作人员，非常感谢你们仔细的校对。

Jeff Forcier
New York，NY
2008年8月

感谢所有围绕在Django、Python和其他Web应用开发的开源基础设施的社区。成百上千热忱的开发人员和维护者的辛勤努力让全世界都能免费用上强大的软件。

我的合作者让我受益匪浅，他们不仅给这项任务带来了必要的知识和技能，还为之做出了很大的贡献。尽管我们分散在美洲的不同角落，我还是有幸见到了Jeff和Wes。

感谢西马萨诸塞州开发组，感谢和你们之间许多非常有趣的geeky的讨论，感谢你们对本书的极大热情。

感谢Hallmark Institute of Photography的主席George J. Rosa III，给了我极大的信任和鼓励，让我选择最好的工具（当然，包括Django）来最好地完成任务。

在2008年夏天经历了一场严重的车祸后，我得到了很多来自家庭、朋友和社区的关心和支持。每一份祝福、每一张卡片、每一分善款和每一顿饭都是很重要的。非常感谢你们。

最后，感谢我亲爱的妻子Kathleen，感谢你的支持、贤惠和爱。

Paul Bissex
Northampton，MA
2008年9月

编写我的第二本书实在是一个非常棒的体验。首先我想要向我的两位出色的合作者致敬，十分高兴能和他们一起工作。他们有汲取他人的Python技巧和介绍Django的经验的能力。我很高兴能合作完成这本出色的Django教程，并且期待能和他们有机会在写作或是教学上再次合作。能编写这样一本完全由开源项目组成的书是非常令人欣慰的，开发人员每天都在使用相同的工具开发改变社会的软件产品。

我要特别感谢Debra Williams Cauley从我第一天加入这个项目开始就帮助我们管理整个进程。虽然在这过程中我们发生了很大的人事变化，但是她一直都让我们专注于写作上面。就像她也赞同的一样，我们的理念是要为社区写出一本“合适的书”，而不仅仅是一本迎合市场需要的Django教程。感谢我们所有技术评论，Michael Thurston（策划编辑），Joe Blaylock和Antonio Cangiano，以及所有在这本书草稿阶段就在Rough Cuts上留下反馈的人，因为你们，这本书才会更加完美。我还要感谢Matt Brown，Django Helper for Google App Engine的首席维护，在审阅附录E中给予的协助，以及Eric Walstad和Eric Evenson最后的终审和评注。

最后，如果没有我们家里团结一致的支持，我是不可能完成这本书的。

Wesley Chun
Silicon Valley，CA
2008年8月

引　言

如果你是一名Web工程师，那么Django有可能彻底改变你的生活。至少它改变了我们。

只要你知道一点构建动态网站是怎么回事的话，那么就一定能体会到不断重复地发明某些标准特性是多么痛苦的一件事情。你得创建数据库结构，把数据导入导出数据库，处理URL，验证用户输入，提供编辑工具，还得关心安全性和可用性……

Web框架的前世今生

终于，你意识到每次都重新实现这些特性实在是太浪费生命了。所以，你决定要重新开发一套自己的库来提供这些功能，或者说，从你最新、最伟大的“创造”中把这些库提取出来。然后，如果要开始一个新项目的话，你第一件要做的事情就是安装你的库。这能大大节约你的工作时间。

但是，事情可没那么简单。如果客户需要的特性不在你的库里怎么办，没关系，加进来就好了。而每个客户都需要不同的东西，结果就变成你在每个服务器上安装的库都有不同的版本。这绝对是没有办法维护的。

有了教训以后，你回过头来把基础库和最好的add-on从各个项目里拿出来重新组合在一起。对绝大多数项目来说你不再需要直接调整库代码，只需要改动一下配置文件就可以了。虽然你的代码库越来越大、越来越复杂，但是它也变得非常强大。

恭喜！你已经完成了一个Web框架。

不过，只要你（或者你的团队、公司或者客户）还在使用它，你就得保证它能正常工作。下次升级操作系统、Web服务器、编程语言的时候会不会把它弄挂了？能不能在未来引入修改时不需要伤筋动骨？它能不能支持复杂但是重要的功能如会话管理、本地化，又或者数据库事务？你的测试覆盖率有多少？

更好的做法

你捧着这本书的原因就是希望能找到更好的做法。要是有一个强大灵活、实现优雅、经历过完整测试而又不要你自己来维护的框架就好了。

你希望用一门真正的编程语言来编写代码，它必须是干净强大、成熟的语言，需要有大量的文档支持。你希望它有一个漂亮的标准库以及大量高品质的第三方库可以满足各种需要，比如生成CSV文件、生成饼状图、科学计算或者是图像处理等。

你需要一个背后有活跃的、助人为乐的用户和开发社区的框架。这个框架能作为一个整体流畅的工作，而其组件又是松散耦合，可以让你在需要的时候随时替换。

一句话，你想要的就是Python和Django。我们编写这本书的目的就是要帮助你尽量简单快

速地在实际环境下学习使用Django。

冲出堪萨斯，走向世界

Django原本是Adrian Holovaty和Simon Willison为堪萨斯州劳伦斯的一家家族媒体World Online编写的。当时是需要它能够迅速开发出数据库驱动的新闻系统。

在Web领域里证明了自己的实力后，Django在2005年7月作为一个开源项目公开发布。搞笑的是，虽然这个时候很多人都觉得Python已经有太多的Web框架了，Django还是迅速获得了大量追随者。今天，Django已经不仅仅是Python框架里的领头羊了，就是与其他所有Web框架里相比也毫不逊色。

当然，World Online还在大量使用Django，它的一些核心工程师还在那里工作。但是自从Django项目开源以来，全世界的公司和组织都已经把它用在了无数大大小小的项目里了。这里有一份不完整的列表：

- The Washington Post
- The Lawrence Journal-World
- Google
- EveryBlock
- Newsvine
- Curse Gaming
- Tabblo
- Pownce

虽然这里还有成千上百的Django网站没有列出，但是Django的传播和成长已经势不可挡，未来会有无数受欢迎的站点采用Django开发。我们希望你也是其中之一。

用Python和Django更好地进行Web开发

Web开发通常是一件很烦琐的工作。你必须要从测试起就面对浏览器的不兼容性、疯狂的爬虫、宽带和服务器的限制以及总体框架等一系列挑战。

虽然我们相信本书将Django的基础讲解得非常完备了，但是我们还要讨论很多其他难点——你80%的时间都会花在这些20%的工作上。我们和许多Django工程师一起工作讨论，听取他们的意见和建议并且帮助他们解决问题。在编写本书的过程中，我们都提醒自己注意这些问题和难点。

要是我们觉得Django和Python不好的话，我们也不会费这么大劲来为它们写一本书。但是当遇到有限制或者有陷阱的时候，我们还是会如实相告的。我们的目标是帮助你完成任务。

目　录

译者序
前言
致谢
引言

第一部分　入门

第1章　Django Python实战 ……1
1.1　Python技术就是Django技术 ……1
1.2　入门：Python交互解释器 ……2
1.3　Python基础 ……3
1.4　Python标准类型 ……5
1.5　流程控制 ……19
1.6　异常处理 ……21
1.7　文件 ……23
1.8　函数 ……24
1.9　面向对象编程 ……33
1.10　正则表达式 ……35
1.11　常见错误 ……36
1.12　代码风格 ……41
1.13　总结 ……43
第2章　Django速成：构建一个Blog ……44
2.1　创建项目 ……44
2.2　运行开发服务器 ……46
2.3　创建Blog应用 ……47
2.4　设计你的Model ……48
2.5　设置数据库 ……48
2.6　设置自动admin应用 ……51
2.7　试用admin ……52
2.8　建立Blog的公共部分 ……55
2.9　最后的润色 ……57
2.10　总结 ……60
第3章　起始 ……61
3.1　动态网站基础 ……61
3.2　理解模型、视图和模板 ……63
3.3　Django架构总览 ……64
3.4　Django的核心理念 ……66
3.5　总结 ……68

第二部分　深入Django

第4章　定义和使用模型 ……69
4.1　定义模型 ……69
4.2　使用模型 ……80
4.3　总结 ……91
第5章　URL、HTTP机制和视图 ……92
5.1　URL ……92
5.2　HTTP建模：请求、响应和中间件 ……96
5.3　视图与逻辑 ……100
5.4　总结 ……105
第6章　模板和表单处理 ……106
6.1　模板 ……106
6.2　表单 ……112
6.3　总结 ……123

第三部分　Django应用实例

第7章　Photo Gallery ……125
7.1　模型 ……126
7.2　准备文件上传 ……127
7.3　安装PIL ……128
7.4　测试ImageField ……128
7.5　构建自定义File变量 ……130

7.6 使用ThumbnailImageField ……134
7.7 设置DRY URL ……135
7.8 Item应用的URL布局 ……137
7.9 用模板把它们都串在一起 ……138
7.10 总结 ……143
第8章 内容管理系统 ……144
8.1 什么是CMS ……144
8.2 Flatpages ……144
8.3 超越Flatpages：一个简单的自定义CMS ……147
8.4 改进建议 ……162
8.5 总结 ……163
第9章 Liveblog ……164
9.1 究竟什么是Ajax ……164
9.2 设计应用程序 ……165
9.3 应用程序布局 ……166
9.4 加入Ajax ……169
9.5 总结 ……176
第10章 Pastebin ……177
10.1 定义模型 ……177
10.2 创建模板 ……179
10.3 设计URL ……180
10.4 试运行一下 ……181
10.5 限制最近Paste显示的数量 ……184
10.6 语法高亮 ……185
10.7 通过Cron Job清除 ……186
10.8 总结 ……187

第四部分 高级Django技术和特性

第11章 高级Django编程 ……189
11.1 自定义Admin ……189
11.2 使用聚合 ……193
11.3 生成下载文件 ……195
11.4 用自定义Manager来增强Django ORM ……200
11.5 扩展模板系统 ……202
11.6 总结 ……211
第12章 高级Django部署 ……212
12.1 编写工具脚本 ……212
12.2 自定义Django codebase ……214
12.3 缓存 ……215
12.4 测试Django应用 ……223
12.5 总结 ……229

附 录

附录A 命令行基础 ……231
附录B 安装运行Django ……240
附录C 实用Django开发工具 ……254
附录D 发现、评估、使用Django应用程序 ……261
附录E 在Google App Engine上使用Django ……264
附录F 参与Django项目 ……273
后记 ……275

清华大学图书馆
C2341291Y

第一部分 入门

第1章 Django Python实战
第2章 Django速成：构建一个Blog
第3章 起始

第1章 Django Python实战

欢迎使用Django和Python！在开始介绍Django之前，我们将会给你介绍一下这个作为Django应用的基石的语言——Python。如果你有其他高级语言编程经验（C/C++、Java、Perl、Ruby等）的话，这一章的内容会比较容易理解。

不过要是你没有任何经验的话也不要紧，Python本身就非常适合作为入门语言来学习。市面上有很多教你如何用Python编程的书。具体你可以参考本章的结尾。我们推荐那些新人先看看这些资源。一旦掌握了基础，回过来再读本章的时候会有更深的体会。

这一章要介绍的Python，主要是语言中和Django开发有关的核心特性和技巧。光有基本的Python技术是无法高效地开发Django的，你还需要知道一些更多Python内部的东西，这样当遭遇到这些特定的特性、这方面的知识和Django的需求时，就不会无所适从。对Python新人或者编程新人来说，先阅读一些其他基本的Python知识以及本章的内容会都可以让你获益良多——选择哪种方式完全可以按照你自己的节奏来。

1.1 Python技术就是Django技术

Django提供了一个高级的框架，用它只需要很少的几行代码就能完成一个Web应用。这种轻盈、强大、灵活的框架让你在设计方案时无需太多考量。Django是以Python写成，Python是一门面对对象的应用程序开发语言，它同时兼具系统语言（如C/C++和Java）的强大和脚本语言（如Ruby和Visual Basic）的灵活迅速。这让Python的用户创建能解决各种疑难杂症。

在这一章里，我们要向你介绍一些我们觉得如果要成为一名出色的Django工程师就必须知道的Python技术。我们主要关注是那些作为Django工程师一定要知道的Python概念，而不是再造一份通用的Python教程。所以基本上这一章里没有什么Django代码。

> Python 2.x还是3.x
>
> 在编写本书的时候，Python正在从2.x向新一代的版本3.0迁移。3.x版本家族不保证对

旧版本的兼容性，所以用2.x编写的代码完全有可能无法在3.x下工作。但是Python的核心开发团队正在努力使这一转换尽可能的顺利：他们会提供可靠的2.x-to-3.x转换工具，并且给予足够的时间保证大家都能顺利完成。

Django的核心团队现在还不打算立刻迁移到3.0上去（和绝大多数面向框架的项目一样，这样的转换破坏性非常大，一定要非常小心），所以这里我们只是提一下这个转换而已。Django只有在绝大多数用户都准备好的时候才会进行升级。

1.2 入门：Python交互解释器

交互解释器是在Python开发过程中最有用的工具之一，有了它就不用去创建、编辑、保存，再运行源文件了，你可以直接用它测试一小段代码。它不但会检查代码的正确性，还能让你在把这段新代码加入到源文件之前进行各种不同的操作，比如查看数据结构、修改键值等。

在阅读本章的过程中，我们建议你启动Python解释器来直接感受一下。大多数Python IDE都能轻易启动它，或者也可以直接从命令行或是你操作系统的应用程序菜单里启动。这样，你就会有一个切身的体会，对Python以及接下来的Django会更有信心。就算是Python高手，哪怕有十多年编程经验的老鸟，也还是每天都离不开Python shell的！

在本书中，你会看到很多用Python shell的交互提示符：>>>打头的代码段。你可以在阅读过程中直接把这些例子输入到解释器里去。像是这样：

```
>>> print 'Hello World!'
Hello World!
>>> 'Hello World!'
'Hello World!'
```

print是非常好用的命令。它不仅向你的用户提供了相关的应用程序信息，还是一个无可替代的调试工具。虽然也有不显示调用print就能“打印”变量值的方法，就像我们前面那样，但是这通常会产生和print不同的结果。

请注意Hello World例子中的不同之处。当直接把一个对象丢给解释器的时候，它会用引号告诉你这是一个字符串。而在使用print语句时，引号则不会显示出来，因为你告诉了解释器去显示字符串的内容（当然是不包括引号的了）。下面这个例子就和字符串的情况有点不同（对数字来说显示都是一样的）。

```
>>> 10
10
>>> print 10
10
```

但是对稍后我们讨论到的复杂对象来说，这个差异就会非常明显，这是因为Python会让你决定在有没有使用print时对象应该如何行为。

虽然后面会详细讲解变量和循环，这里我们先来体验一下稍微复杂一点的Python是什么样的。这里有几个for循环。

```
>>> for word in ['capitalize', 'these', 'words']:
...     print word.upper()
```

```
...
CAPITALIZE
THESE
WORDS
>>> for i in range(0, 5):
...     print i
...
0
1
2
3
4
```

Python语法里非常重要的一个方面就是没有区分代码块的花括号（{}）。我们用对齐来代替括号：对于给定的一段Python代码，它们必须有相同的对齐，通常是4个空格（任意数目的空格或者tab也都可以）。如果你习惯于其他语言的话，这可能需要一点时间来适应。但是过一段时间以后，你就会发现其实这种方式也没有看上去那么糟糕。

关于解释器最后一点要说的就是：当你熟练使用解释器了以后，你应该考虑试试另一个类似的工具IPython。要是你喜欢交互解释器的话，IPython的能力完全是在另一个数量级上的哦！它提供了系统shell访问、命令行记数、自动对齐、命令行历史等许多特性。你可以在http://ipython.scipy.org上找到更多关于IPython的信息。它不是随Python一起发布，而是一个第三方应用。

在解释器里使用Django

要是能在Python解释器里调试Django应用或框架的话那就太方便了。但是如果你只是按照常规启动解释器并且试着导入Django模块的话，你只能得到一个DJANGO_SETTINGS_MODULE没有设置的错误。为了方便起见，Django提供了一个manage.py shell命令，它能进行一些必要的环境设置来避免这个问题。

如果你安装了iPython的话，manage.py shell会默认使用它。如果你安装了iPython，但是又希望使用标准Python解释器的话，可以运行manage.py shell plain。虽然在后面的例子里我们是继续使用默认的解释器，不过我们还是强烈推荐iPython。

1.3 Python基础

在这一节里我们要介绍Python基本的几个方面，包括注释、变量、运算符和基本Python类型。下面几节里会更详细地介绍Python的主要类型。绝大多数Python（和Django）代码都是保存为以.py扩展名为结尾的文本文件，这是告诉系统一个Python文件的标准做法。你还会看到相关的扩展名诸如.pyc或者.pyo，这不会影响你的使用，现在可以暂时先忽略它们。

注释

Python的注释是用井字符（#）表示的。如果它是一行里第一个字符，那么这整个一行就

都是注释。#还可以出现在行的中间，这代表从井字符开始的地方起，同一行中剩下的部分都属于注释。例如：

```
# this entire line is a comment
foo = 1          # short comment: assign int 1 to 'foo'
print 'Python and %s are number %d' % ('Django', foo)
```

注释不仅可以用来解释邻近的代码，还可以组织代码运行。一个很好的例子就是Django的settings.py配置文件——注释去掉了不必要的常用选项，或者是不用于默认值的值，要重新启用它们也很简单，这样配置选项也变得可见了起来。

变量和赋值

Python的变量不像其他语言那样需要先为之声明一个特定的类型。Python是一门“动态类型”的语言。我们可以把变量想象成指向一个匿名对象的名字，这个对象才真正地保存着实际的值。所以，变量的值可以在任何时刻发生改变，像这样：

```
>>> foo = 'bar'
>>> foo
'bar'
>>> foo = 1
>>> foo
1
```

在这个例子里，变量foo首先被映射到一个字符串对象'bar'上，接着再映射到另一个整数对象1上去。注意，除非有其他变量也引用了foo先前引用的那个字符串（这是完全可能的！），否则它就彻底消失了。

因为你可以像这样随时改变变量名的映射，除非你向解释器询问更多信息，否则你不可能百分之百地确定变量在任何给定的时刻指向的是什么类型的对象。不过，只要这个变量能够表现的像个类型的话（比如，它有一个字符串应该有所有方法），它就可以被认作为那个类型，不管它是不是有额外的属性。这叫做“duck-typing”，即，只要它走起来像只鸭子，叫起来也像只鸭子的话，我们就可以把它当作一只鸭子。

运算符

说到运算符，基本上和你熟悉的其他编程语言所支持的运算符一样。这包括算术运算符，如+、-、*等，以及它们的相应的赋值运算符+=、-=、*=等。就是说x = x + 1和x += 1是一样的。不过这不包括其他语言里有的自增/自减运算符（++和--）。

标准的比较运算符，如<、>=、==、!=等也都是支持的，你还可以分别用布尔运算符and和or将它们组合起来。Python还支持取反的布尔运算符not。下面是一些例子：

```
show_output = True
if show_output and foo == 1:
    print 'Python and %s are number %d' % ('Django', foo)
```

你已经知道了Python的代码段是用对齐而不是花括号来区分的了。我们前面提到这样就能

非常容易地辨别某块代码段是属于哪里的，更进一步来说，我们可以消除所谓的“else悬挂”(dangling-else) 问题，因为一个else子句只能属于其中一个if，绝对没有歧义产生。

同样的，Python也避免使用很多符号。不仅是那些花括号，Python没有结尾的分号（;），没有货币符号（$），在条件语句里也不需要小括号（()）。你可能注意到了一些“@”符号和大量的下画线（_），但是这就是仅有的符号了。Python的作者相信只有干净和容易阅读的代码才能避免混乱。

1.4 Python标准类型

现在我们来介绍作为Django程序员会用到的标准类型。它们可以是标量（scalar）或者文字量（literal），(比如数字和字符串)。也可以是用来将多个Python对象组织在一起的“容器”，或者说数据结构。在我们开始介绍主要数据类型之前，首先要注意到的是所有Python对象都有内在的布尔值。

对象的布尔值

和大多数语言一样，Python可以表示两种布尔值：真（True）和假（False）。无论其本身的数据值是什么，所有Python值都可以表示为布尔值。举例来说，任何等于零的数值都被认为是False，而所有非零的数值则为True。类似的，空的容器即为False，非空的容器就为True。

你可以用bool函数来决定任何Python对象的布尔值，甚至True和False它们自己也是合法的值，可以被当作变量值来显式地赋值。

```
>>> download_complete = False
>>> bool(download_complete)
False
>>> bool(-1.23)
True
>>> bool(0.0)
False
>>> bool("")
False
>>> bool([None, 0])
True
```

上面的例子里所有bool的输出都是有意义的。就是最后一个例子有点绕：虽然列表的两个元素都是False值，但是这个非空的列表却有一个True值。在Python的if和while语句里做条件判断决定是不是要执行代码时就要依赖于这些值来判定对象的布尔值。

注意最后那个例子里的None值。这个Python的特殊值和其他语言里的NULL或者void是一样的意思。None在做布尔判断时总是为False。

布尔值和数字一样是文字量。下面我们来看Python中的数字。

数字

Python有两个主要的数值类型：int（整数）和float（浮点数）。根据KISS原则，Python只

有一种整数类型int，而不是像其他语言那样提供了好几种整数类型[⊖]。除了十进制整数还可以表示为十六进制数和八进制数。浮点数就是和其他语言一样的双精度的浮点实数。下面就是一些整数和浮点数的例子，以及在解释器里的使用方法：

```
>>> 1.25 + 2.5
3.75
>>> -9 - 4
-13
>>> 1.1
1.1000000000000001
```

嗯……最后一个例子有点奇怪。浮点数的范围很大，但是，它们在表示有理数的时候不是非常精确。故而还有另一个浮点数类型Decimal（这不是一个内置类型，必须通过decimal模块访问），它的值范围比较小，但是更精确。Python还为科学计算提供了一个内置的复数类型。

表1.1总结了这些数字类型，并且给出了更多的例子。

表1.1　Python内置数字类型

类　型	描　述	举　例
int	带符号整数（没有大小限制）	−1，0，0xE8C6，0377，42
float	双精度浮点数	1.25，4.3e+2，−5，−9.3e，0.375
complex	复数（实部+虚部）	2+2j，.3−j，−10.3e+5−60j

数字运算符

数字支持大多数常用的算术运算符：加（+）、减（−）、乘（*）、除（/和//）、取模（%）和指数运算（**）。

如果两个操作数都是整数的话，除法运算符 / 代表“经典除法classic division”，就是说它会截断结果（即向下取整floor division），而对浮点数来说则是“真正的除法true division”。Python还提供了显式的“向下取整的除法floor division”，不管操作数的类型是什么，一律都返回整数结果：

```
>>> 1 / 2          # floor division (int operands)
0
>>> 1.0 / 2.0      # true division (float operands)
0.5
>>> 1 // 2         # floor division (// operator)
0
>>> 1.0 // 2.0     # floor division (// operator)
0.0
```

最后，Python可以对整数进行二进制位操作，与（&）、或（|）、异或（^）、取反（~）和左右移位（<<和>>），以及对它们进行赋值运算，如&=，<<=等。

⊖ Python曾经还有另一个整数类型long，但是它现在已经被合并到int里来了。在旧有的代码和文档里你还能看到用'L'字符结尾所表示的长整数。像1L，−42L，999999999999999L等。

内置数字工厂函数

每种数字类型都有一个工厂函数让用户能在各种数字类型之间相互转换。有的读者把这叫做“conversion”或者“casting”，但是在Python里我们不用这些个术语，因为你并没有真的“改变”这个现有对象的类型。实际上你是在原来那个对象的基础上返回了一个新的对象（所以这里用“factory”）。即，int(12.34)会创建一个新的值为12的整数对象，而float(12)则会返回12.0。complex和bool函数也是一样的道理。

Python还提供了很多内置函数用以操作数字，像round函数可以将浮点数精确到指定的某一位，而abs函数则可以取得一个数的绝对值。下面是一些使用这些函数的例子：

```
>>> int('123')
123
>>> int(45.67)
45
>>> round(1.15, 1)
1.2
>>> float(10)
10.0
>>> divmod(15, 6)
(2, 3)
>>> ord('a')
97
>>> chr(65)
'A'
```

要知道更多关于这些函数的信息，可以参考《Core Python Programming》（Prentice Hall, 2006）中的“数字”章节，或是查阅任何完整的参考书籍，也可以在网上搜索Python的文档。现在我们来看看字符串和Python重要的容器类型。

序列和迭代

很多编程语言都将数组作为数据结构，数组通常是定长的，由一组相似的对象组合而成，可以通过下标顺序访问。Python的序列类型基本上也是一样的意思，但是它可以包含不同类型的对象，同时其长度还可以长短伸缩。在这一节里，我们讨论两个十分常用的Python类型：列表list（[1,2,3]）和字符串string（'python'）。他们都属于序列sequence这种数据结构的一种。

序列是Python类型里很典型的可以迭代的类型：即这是一种你可以让你每次获取一个元素的类型。迭代背后的基本思想是你可以用next方法持续地要求下一个对象，而它则不停地“读出”其内部集合里的元素直至耗尽为止。Python序列不但支持这种方式的迭代（虽然99%的情况下你都是用for循环而不是next方法），它还支持随机访问——即你可以直接要求序列中某个位置的对象。例如，my_list[2]就会返回列表中第三个元素（下标从0开始）。

第三种序列类型是元组tuple。基本上它就是一个“身有残疾的只读列表”，除此之外它也没什么特别的地方了——当然它的作用是完全不同的。虽然它不是你应用数据结构的第一选择，不过我们还是需要告诉你它的含义和作用。鉴于你应该早就知道字符串是什么了，我们将会从列表开始，然后再介绍元组。表1.2列出了我们要讨论的类型并且给出了几个例子。

表1.2 序列类型举例

类 型	举 例
str	'django', '\n', "", "%s is number %d" % ('Python', 1), """hey there"""
list	[123, 'foo', 3.14159], [], [x.upper() for x in words]
tuple	(456, 2.71828), (), ('need a comma even with just 1 item',)

序列切片

刚才我们提到了序列是可以直接索引的，接下来是一些操作字符串的例子。和许多其他语言不同，Python的字符串可以同时被当作是离散的对象和一组单独的字母。

```
>>> s = 'Python'
>>> s[0]
'P'
>>> s[4]
'o'
>>> s[-1]
'n'
```

Python还提供非常灵活的负索引。我们一定都写过类似这样的代码data[len(data) - 1]或是data[data.length - 1]来获取数组中最后一个元素吧？在上面代码里的最后一个例子中，我们只需一个简单的-1就可以了。

你还可以一次索引序列中的多个元素，在Python中称之为切片（slicing）。切片是用一组用冒号（:）隔开的索引下标来表示的，比如*i*和*j*。当要求切片的时候，解释器会取以第一个下标*i*为起始，到第二个下标*j*为结束（但是不包括j）的子集。

```
>>> s = 'Python'
>>> s[1:4]
'yth'
>>> s[2:4]
'th'
>>> s[:4]
'Pyth'
>>> s[3:]
'hon'
>>> s[3:-1]
'ho'
>>> s[:]
'Python'
>>> str(s)
'Python'
```

如果缺了一个下标，意思就是根据缺了哪个下标，分别代表从哪里开始或到哪里结束。这里[:]是一个非常重要的切片（它代表了返回整个序列的一个拷贝[⊖]）。最后，虽然我们前面的例子都是作用于字符串之上，但是切片语法对列表和所有其他序列类型也都是有效的。

⊖ 这里所谓的“拷贝”，是指引用的拷贝，而不是对象本身。更准确的说法是“浅拷贝”。参见下面章节可以获取更多关于对象复制的内容。

其他序列操作符

在上一节我们看到了用[]和[:]进行切片操作的方法。还有其他一些可以作用于序列之上的操作包括了连结（+）、复制（*）以及检查是否是成员（in和not in）。和前面一样，这里用字符串来举例，当然这些也是能用于其他序列之上的。

```
>>> 'Python and' + 'Django are cool!'
'Python andDjango are cool!'
>>> 'Python and' + ' ' + 'Django are cool!'
'Python and Django are cool!'
>>> '-' * 40
'----------------------------------------'
>>> 'an' in 'Django'
True
>>> 'xyz' not in 'Django'
True
```

连结的其他形式

我们建议你避免在序列上使用 + 操作符。当你还是新手的时候，它确实能很方便地将字符串组合在一起。但是这样做的效率不高。（细节的原因需要解释Python下C的实现手法，这不是本书的内容——你只要相信我们就好了。）

例如拿字符串来说，你可以用稍后字符串一节中要讨论的字符串格式化操作符（%）来取代'foo'+'bar'，像这样'%s%s' % ('foo', 'bar')。另一种办法用join方法，特别是当要把一组字符串拼在一起的时候非常方便，例如''.join(['foo', 'bar'])。对于列表来说，extend方法也可以把另一个列表中的内容加进来（相比list1 += list2，list1.extend(list2)要好得多）。

列表

Python类型里很像其他语言中数组的一种类型是列表list。列表是可变的，可以改变大小的序列，它能够保存任何数据类型。下面的例子里我们展示了如何创建一个列表，以及你能进行的操作。

```
>>> book = ['Python', 'Development', 8]      # 1) 创建列表
>>> book.append(2008)                        # 2) 附加对象
>>> book.insert(1, 'Web')                    # 3) 插入对象
>>> book
['Python', 'Web', 'Development', 8, 2008]
>>> book[:3]                                 # 4) 切片头三个元素
['Python', 'Web', 'Development']
>>> 'Django' in book                         # 5) 对象属于列表吗?
False
>>> book.remove(8)                           # 6) 显式移除对象
>>> book.pop(-1)                             # 7) 通过索引移除对象
2008
>>> book
['Python', 'Web', 'Development']
>>> book * 2                                 # 8) 重复/复制
```

```
['Python', 'Web', 'Development', 'Python', 'Web', 'Development']
>>> book.extend(['with', 'Django'])          # 9) 合并到当前列表
>>> book
['Python', 'Web', 'Development', 'with', 'Django']
```

我们来逐条解释一下上面的例子：

1. 创建一个初始有两个字符串和一个整数的列表。
2. 在列表尾部添加另一个整数。
3. 在第二个位置上插入一个字符串（下标为1）。
4. 获取头三个元素的一个切片。
5. 成员检查（元素是否属于列表）。
6. 无论元素的位置，从列表中移除它。
7. 根据位置（即下标）移除（并返回）一个元素。
8. 展示复制操作符 * 的用法。
9. 用另一个列表扩展本列表。

如你所见，列表是一种非常灵活的对象。下面我们再深入讨论一下它的方法。

列表的方法

我们先重设前面例子中的列表，并对其进行排序，然后再讨论其细节。

```
>>> book = ['Python', 'Web', 'Development', 8, 2008]
>>> book.sort()                  #注意：这是直接排序……，没有返回值!
>>> book
[8, 2008, 'Development', 'Python', 'Web']
```

对一个混合类型的列表进行“排序”的结果其实是未定义的。没有关系的对象（比如字符串和数字）怎么能相互比较呢？Python所用的算法是“尽量猜测”来获取“正确的行为”：先对所有数字值进行排序（从小到大），然后再对字符串按字典序排序。这个例子还算有点意义，不过要是你把文件或者是类实例放进去的话，那结果就一定是未定义的了。

列表的内置函数如sort、append和insert等都是直接对对象进行修改而且没有返回值。Python的新手可能会觉得sort不返回一个排好序的列表的行为相当奇怪，所以使用的时候一定要小心。另一方面，之前看到的字符串方法upper却返回了一个字符串（包含了一个全部都是大写的字符串拷贝）。这是因为和列表不同，字符串是不可改变的，所以upper方法只能返回一个（修改过的）拷贝。稍后会详细介绍可改变性。

当然，有时候我们确实希望获得一个排好序的拷贝而不是直接在给定序列上排序。Python 2.4以上的版本提供了内置函数sorted和reversed，它们分别接受一个列表作为参数并且返回一个排好序的或者是倒序的拷贝。

列表推导式

列表推导式（list comprehension）是一个由逻辑代码组成的结构（construct，这是从另一个语言Haskell里借鉴来的），它构造了一个包含了由那段逻辑代码所生成的值或对象的列表。例如，有一个包含整数0～9的列表。如果我们要对其中的每一个数字加一，并且返回的结果也要

是一个列表的话应该怎么做呢？有了列表推导式（或者简写为“listcomps”），我们可以这样：

```
>>> data = [x + 1 for x in range(10)]
>>> data
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

和列表一样，列表推导式也采用方括号表示，并且用到了一个简写版的Python for循环。虽然我们还没有讲到循环（马上就会介绍），但是你还是应该很容易理解这个推导式。第一部分是一个生成结果列表元素的表达式，第二部分是一个输入表达式（input expression，会生成一个序列）上的循环。

阅读理解列表推导式的推荐做法是先从里面的for循环开始，向右查看是否有if条件（上面这个例子里没有），然后将推导式开始的表达式映射到每一个匹配的元素上去。看看下面这个例子。

```
>>> even_numbers = [x for x in range(10) if x % 2 == 0]
>>> even_numbers
[0, 2, 4, 6, 8]
```

第二个例子展示了如何使用最后的if子句来过滤元素。它不包含任何做出修改的逻辑代码——x自身也是一个合法的表达式（结果就是x的值）。在过滤序列时列表推导式非常好用。

生成器表达式

Python还支持另一种和列表推导式类似的结构，叫做生成器表达式（generator expression）。除了它有一种称之为“惰性计算”（lazy evaluation）的特点，它和列表推导式的用法基本一致。它工作的方式是每次处理一个对象，而不是一口气处理和构造整个数据结构，这样做的潜在优点是可以节省大量的内存（虽然有时候要付出一点性能上的代价）。

在上一个例子中，我们用列表推导式在10个元素的列表中查找偶数，但如果列表中有一万个甚至上千万个数字时要怎么做？要是列表中不是简单的整数，而是复杂庞大的数据结构时又怎么办？在这种情况下，生成器表达式节省内存的特点就能帮助我们解决问题了。我们只要稍微修改一下语法，将方括号替换成小括号就能把列表推导式变成生成器表达式（genexp）了。

```
>>> even_numbers = (x for x in range(10000) if x % 2 == 0)
>>> even_numbers
<generator object at 0x ...>
```

生成器表达式是Python 2.4以后才支持的特性，所以如果你还在使用Python 2.3的话，就没办法使用它了，目前它在Python程序员之间的名气也还不是很响亮。不过，要是你的输入序列可能会变得很大时，最好考虑采用生成器表达式而不是列表推导式。

字符串

Python的另一个序列类型是字符串，虽然它们都是用单引号或者是双引号括起来的（'this is a string'或是"this is a string"），但是你可以把它们想象成一个字符数组。另外，和列表不同的是，字符串是不能修改的，其大小也不能改变。任何试图改变字符串长度或是修改内容的行

为实际上只是创建了一个新的修改过的字符串而已。不过对普通使用来说，这些操作都是透明的，只有在处理内存问题时才会显得重要。

和列表一样，字符串也有很多方法，这里要再次重申，因为字符串是不可改变的，字符串没有一个能修改现有的对象，它们只是返回了一个修改过的拷贝而已。在编写本书时，字符串已经有超过37个方法了！我们主要介绍一些可能会在Django里用到的方法。下面是一些例子：

```
>>> s = 'Django is cool'                    # 1
>>> words = s.split()                       # 2
>>> words
['Django', 'is', 'cool']
>>> ' '.join(words)                         # 3
'Django is cool'
>>> '::'.join(words)                        # 4
'Django::is::cool'
>>> ''.join(words)                          # 5
'Djangoiscool'
>>> s.upper()                               # 6
'DJANGO IS COOL'
>>> s.upper().isupper()                     # 7
True
>>> s.title()                               # 8
'Django Is Cool'
>>> s.capitalize()                          # 9
'Django is cool'
>>> s.count('o')                            # 10
3
>>> s.find('go')                            # 11
4
>>> s.find('xxx')                           # 11
-1
>>> s.startswith('Python')                  # 12
False
>>> s.replace('Django', 'Python')           # 13
 Python is cool'
```

我们来小结一下上面的例子：

1. 创建字符串。

2. 用空白符把字符串分隔到列表里去。

3. #2的逆操作。（把子串列表组合成一个字符串并以空格分隔。）

4. 和#3一样，但是以一对冒号分隔。

5. 和#3一样，但是没有分隔符。（将所有子串合并在一起。）

6. 创建（并且丢弃）一个新字符串，在原有版本上全部大写。（另外还可以参考lower方法。）

7. 展示了如何串联函数调用来确认一个字符串是不是全部大写。（另外还可以参考islower等方法。）

8. 让字符串的每个单词都以大写开头，剩下的部分都是小写。

9. 只大写第一个单词的开头，其他部分小写。

10. 计算子串'o'在字符串里出现的次数。

11. 找出子串'go'在字符串中的起始下标（索引值为4）。（还可以看考index方法。）

12. 和#10一样，但是没有找到匹配，所以返回-1。

13. 检查字符串是不是以给定子串开头。在这里为否。（另外请参考endswith方法）。

14. 简单的查找替换。

和split方法类似的，还有一个splitlines方法，它会特别地去查找行尾符（而不是空白符）。如果你有一个包含这些字符的字符串，比如从文件中读取数行的时候，你可以用rstrip方法将所有结尾的空白符号全部去掉（或者strip方法也行，它会移除行首行尾的空白符）。

例子#7展示了只要你明确知道每一个方法调用所返回的对象是什么类型，就可以将函数串联起来使用。因为我们知道upper方法返回的是字符串，所以完全可以在新的字符串上直接调用另一个字符串方法。在这个例子里，我们调用了isupper方法来确认一个字符串是不是全部由大写字母组成。如果返回的对象是一个列表的话，那么你就可以调用列表的方法。

除了isupper方法，字符串还有很多其他以is-开头的方法，例如isalnum、isalpha等。表1.3总结在本节出现过的方法。和你推测的一样，字符串还有很多其他方法，我们推荐你查阅Python的文档来获取更多关于字符串方法的内容。

表1.3 常用的Python字符串方法

字符串方法	描 述
count	字符串中子串出现的次数
find	查找子串（类似的还有index、rfind、rindex）
join	用一种分隔符合并子串
replace	查找替换子串
split	将字符串分隔成子串（类似的还有splitlines）
startswith	字符串是不是以子串开头（另请参见endswith）
strip	移除行首行尾的空白符（另请参见rstrip、lstrip）
title	大写每个单词第一个字母（另请参见capitalize、swapcase）
upper	大写整个字符串（类似的有lower）
isupper	字符串是大写的么？（类似的还有islower等）

字符串指示符

Python字符串可以在引号前放一个标识符：r代表这是一个raw字符串，而u则代表这是一个Unicode字符串。这些指示符（designator）在编写代码和在解释器里显示字符串时都用的着。

```
>>> mystring = u'This is Unicode!'
>>> mystring
u'This is Unicode!'
```

但是打印和转换raw或Unicode字符串的操作却不会打印指示符：

```
>>> mystring = u'This is Unicode!'
>>> print mystring
```

```
This is Unicode!
>>> str(mystring)
'This is Unicode!'
```

“raw”指示符告诉解释器不要转换字符串中的任何特殊字符。例如，特殊字符\n通常代表一个新行，不过有时候你不希望发生类似这样的转换，比如一个DOS的文件名：

```
filename = r'C:\temp\newfolder\robots.txt'。
```

另一个使用raw字符串的地方是正则表达式，因为它大量使用了诸如反斜杠（\）这样的特殊字符。在正则表达式一节中，我们会用raw字符串来表示一个正则表达式，r"\w+@\w\.\w+"，这比普通的（需要转义符的）字符串要容易阅读多了，"\\w+@\\w+\\.\\w+"。

Django中的raw字符串

在使用正则表达式的Python代码中，raw字符串是十分常见的。拿Django来说，就是在你URL的配置规则中，Django根据匹配URL请求和你提供的正则表达式的结果来控制分派。用raw字符串来编写这些规则能让它们看上去更清晰一些。而且为了一致性，不管一个正则表达式是否用到了反斜杠，通常一律都使用raw字符串。

因为通常普通的Python字符串只含有很少的一些字符（包括Western字母加上一些特殊符号），它们不足以显示很多非英语语言的字符。Unicode是一种新型的字符集，拥有大量的编码方式，所以不存在这个限制。Django（在编写本书时）已经花费了相当多的经历来保证框架的每个部分都能支持Unicode。因此在开发Django的过程中，你会遇到很多Unicode字符串对象。

字符串格式化操作符和三引号字符串

在本章之前的例子里，已经见过字符串格式化操作符（%）了。它通过“格式化字符串”（format string）来为将不同的输入类型打印到字符串做准备，这个格式化字符串包含了一些特殊的转换说明（directive）。这里是一些例子。

```
>>> '%s is number %d' % (s[:6], 1)
'Django is number 1'
>>> hi = '''hi
there'''
>>> hi
'hi\nthere'
>>> print hi
hi
there
```

在上面的例子中，我们有一个字符串（其格式化字符串的转换说明是%s）和一个整数（%d）。格式化操作符将它左边的格式化字符串和右边的参数元组（不是列表）组合起来。这些参数必须和转换说明一一对应。你可以参考Python文档以获得所有可用的格式化转换说明。

上面的例子还展示了三引号的使用，这是Python独此一家的特性。你可以在字符串中插入任何特殊字符。如果你要生成一个相当长的字符串，而又不希望考虑诸如\\n或者\\r\\n这样的行尾符来保证折行的话，使用三引号就可以了，像下面这样的XML代码段。

```
xml = '''
<?xml version="1.0"?>
<Request version="%.1f">
    <Header>
        <APIName>PWDDapp</APIName>
        <APIPassword>youllneverguess</APIPassword>
    </Header>
    <Data>
        <Payload>%s</Payload>
        <Timestamp>%s</Timestamp>
    </Data>
</Request>
'''
```

最后，注意上面例子中虽然用到格式化转换说明，但是却没有格式化操作符和相对应的元组。这是因为字符串格式化操作符就是一个操作符而已。你完全可以先在代码里定义好格式化字符串，稍后再用操作符和元组填充之。

```
import time          # use time.ctime() for timestamp
VERSION = 1.2        # set application version number

# [...]

def sendXML(data):  # define some sendXML() function
    'sendXML() - transmit XML data'

# [...]
payload = 'super top-secret information'
sendXML(xml % (VERSION, payload, time.ctime()))
```

元组

元组（tuple）是前面讨论的列表的近亲。表面上两者的一个明显区别在于列表用方括号表示而元组用小括号，不过更深层次的区别就又要论及Python的对象模型了。虽然列表允许并且提供了方法来改变它的值，但元组是不可改变的，即它不允许你改变它的值——这也是它们没有方法的部分原因。

乍看之下，新学Python的程序员会很奇怪为什么要有这么一个单独的数据类型，或者说，干嘛不直接提供一个“只读”的列表呢？看起来，这个理由满充分的，但实际上，元组提供的功能远比一个“只读”列表要多的多。它们的主要作用是作为参数传递给函数调用、或是从函数调用那里获得参数时，保护其内容不被外部接口修改。

这不是说它们对程序员就没有用了。虽然它们在前台上的用武之地不大，但是在后台却是使用得相当频繁的。例如在Django的配置文件中就会见到大量的元组。虽然没有方法，但是它们依然进行常用的序列操作，也可以用在内置函数里。

Django中容易犯的和元组相关的错误

在典型的Django应用里你经常会看到元组的使用，对Python的新手来说，这或许是最晦涩

的序列类型了——特别是单个元素的元组要求在最后“必须”跟一个逗号的时候。下面我们来看看你能不能理解下面的这些例子：

```
>>> a = ("one", "two")
>>> a[0]
'one'
>>> b = ("just-one")
>>> b[0]
'j'
>>> c = ("just-one",)
>>> c[0]
'just-one'
>>> d = "just-one",
>>> d[0]
'just-one'
```

第二个例子看着有点问题，怎么回事呢？记住，元组是由逗号决定的，而不是小括号。所以b在这里只是一个字符串而已，这就是为什么b[0]返回的是字符串的第一个字母。而小括号的最后一个逗号则决定了那是一个元组，所以c[0]返回的正是我们希望的结果。我们甚至可以整个去掉小括号，就像上面例子中的d一样——虽然通常这样做不太好。Python的宗旨之一就是尽量显式地表达意思而不要依赖于默认行为。

很多Django的配置都是用元组来指定的——admin选项、URLconf规则，以及很多在settings.py中的设置。所以Django的有些部分和其他部分比较起来更容易告诉你错误的内容。如果你在设置admin选项时用了字符串而不是元组，那么你会得到一个很有用的错误信息，例如告诉你"admin.list_display"必须是一个列表或是元组。另一方面，如果你的ADMINS或MANAGERS设置缺少了结尾的逗号，你会发现服务器会试图给你名字里的每个字母发送错误提示的email！因为这是一个在Django的新手中非常容易犯的错误，我们会在下面的“常见错误”一节中再次强调。

内置序列函数和工厂函数

与数字一样，所有序列类型都有一个特殊的工厂函数来复杂创建所需类型的实例：list、tuple和str，另外还有一个为Unicode字符串准备的unicode函数。通常str负责为一个对象提供可读的或者说“可打印的”（printable）字符串表示。在Python里的另外一个函数repr也提供相似的功能，但是它产生的是一个“可计算的”（evaluatable）字符串表示。这通常意味着，它是一个将Python对象作为字符串的纯表示（pure representation），如果对这个字符串运行eval语句的话应该可以把它再转成对象。

内置函数len能告诉你序列里有多少个元素。max和min分别返回序列中“最大”和“最小”的对象。any和all告诉你序列中是否“任意”或“全部”元素都是True。

在其他语言里的for循环需要自己计数，而籍由range函数的帮助，Python中的for循环则更具有迭代的风格。不过，内置函数enumerate则把这两种风格结合了起来。它返回了一个特殊的迭代器，既能访问序列的下标，同时又能访问其相应的元素。

在表1.4中我们列出了所有这些常用的序列函数。

表1.4 内置序列函数和工厂函数

函数	描述[1]
str	(可打印的) 字符串表示 (参见repr, unicode)
list	列表表示
tuple	元组表示
len	对象的势
max	序列中“最大的”对象 (参见min)
range	给定范围里可迭代的数字 (参见enumerate, xrange)
sorted	返回排好序的列表 (参见reversed)
sum	序列值加和 (数字)
any	是不是有元素为True? (参见all)
zip[2]	返回N个元组的迭代器，其中每个元组包含了N个序列里对应的元素

①虽然这里很多函数的描述都标为“序列”。但是实际上这些函数可以用在所有“可迭代”对象上。这里说的可迭代对象是指任何类似序列的数据结构，只要它可以迭代就行，比如序列、迭代器、生成器、字典的键值和文件里的行等。

②这里的描述不是很清楚，有点拗口。zip函数接受多个序列，然后将每个序列对应位置的元素组织在一起组成一个新的元组，最后返回由这些元组组成的列表。它的长度由参数序列中最短的长度决定。举个简单的例子：

```
>>> zip ([1, 2, 3], ('a', 'b', 'c'))
[(1, 'a'), (2, 'b'), (3, 'c')]
```

映射类型：字典

字典是Python里唯一的映射类型。字典是可变的、无序的、大小可变的键值映射，有时候也称为散列表 (hashes) 或是关联数组。虽然其语法和序列很相似，但你是用键而不是下标来访问元素值，而且字典用花括号 ({}) 而不是方括号 (列表) 或是小括号 (元组) 来定义的。

字典是目前讲到的Python语言里最重要的数据结构。它是大多数Python对象的幕后黑手。不管对象是什么类型或怎么使用的，绝大多数情况下在幕后都有一个字典在管理着它的属性。闲话少说，我们来看看究竟什么是字典以及它能干什么。

字典的操作、方法和映射函数

下面是一些如何使用字典的例子。我们来讨论一下这些代码的作用以及给出这些操作符、方法和函数的用法。

```
>>> book = { 'title': 'Python Web Development', 'year': 2008 }
>>> book
{'year': 2008, 'title': 'Python Web Development'}
>>> 'year' in book
True
>>> 'pub' in book
False
>>> book.get('pub', 'N/A')          # where book['pub'] would get an error
'N/A'
>>> book['pub'] = 'Addison Wesley'
>>> book.get('pub', 'N/A')          # no error for book['pub'] now
'Addison Wesley'
>>> for key in book:
```

```
...     print key, ':', book[key]
...
year : 2008
pub : Addison Wesley
title : Python Web Development
```

那么上面的代码都做了些什么？我们来总结一下：

1. 创建一个初始字典，它包含有一个字符串和一个整数。它们的键都是字符串。
2. 显示这个对象。
3. 检查字典的时候含有某个键（两次，一次为真，一次为假）。
4. 使用get方法获取给定键的值（在这里获取的是默认值）。
5. 加入一个新的键-值对。
6. 再次使用get方法，不过这次成功地获取了值。
7. 迭代整个字典并显示每一对键-值。

我们先来看看最后一段代码。我们用了一个for循环来迭代字典里所有键。这是非常典型的手法。你还可以确认我们刚刚说到的特点：字典的键是无序的（如果我们关心顺序的话，我们就会用序列！），而且正是这种无序让字典在查找元素值的时候非常地高效。

回到第4步，如果传递给get的键不存在于字典的话，它就会返回第二个参数（如果你没有设置这样一个默认值的话就会返回None），你也可以用方括号语法，就像从序列里获取一个元组一样，d['pub']。这里的区别是'pub'不是字典的一个成员，所以d['pub']会返回一个错误。相比下来get方法更加安全，因为它总是返回一个值而不会发生错误。

另一个更加强大的方法是setdefault。它的作用和get一样，而且要是键不存在的话，它还会用默认值自动创建键-值对，这样当后面再调用“dict.get(key)”或“dict[key]”时，它们就不会产生错误了。

```
>>> d = { 'title': 'Python Web Development', 'year': 2008 }
>>> d.setdefault('pub', 'Addison Wesley')
'Addison Wesley'
>>> d
{'year': 2008, 'pub': 'Addison Wesley', 'title': 'Python Web Development'}
```

如果你早就知道散列表的话，那么你一定对“键冲突”（key collision）这个概念很熟悉了。这是当你试图用一个已经存在的键往散列表里添加另一个值时会发生的事情。在Python里没有这种情况，所以如果你把另一个对象用一个已经存在的键赋值的话，它会覆盖前一个值。下面我们来演示如何用del关键字来移除键-值对，让字典回到之前开始的状态：

```
>>> del d['pub']
>>> d['title'] = 'Python Web Development with Django'
>>> d
{'year': 2008, 'title': 'Python Web Development with Django'}
>>> len(d)
2
```

第一行用del命令移除了一个键-值对。随后我们给title键赋予了另一个字符串值，把之前的那个替换掉。最后，我们用通用的内置函数len获取字典的长度，即字典里多少键-值对。表1.5小结了常用的字典方法。

表1.5 常用的Python字典方法

字典方法	描 述
keys	所有键（另请参见iterkeys）
values	所有值（另请参见itervalues）
items	所有键-值对（另请参见iteritems）
get	获取给定键的值或默认值（另请参见setdefault、fromkeys）
pop	从字典里移除键并返回值（另请参见clear、popitem）
update	用另一个字典更新字典

Django中“类似字典的”数据类型

字典（或“dict”）是有Python内建支持的，所以很自然地Django会大量地使用它。然而，有的地方我们不仅需要字典这样的键-值行为（key-value behavior），还需要一些字典没有提供的功能。一个最显著的例子就是HttpRequest对象中保存GET和POST参数的QueryObject对象。由于为同一个给定的参数（字典的键）提交多个值在这里是合法的，但是Python的字典却无法提供这个功能，我们需要一个特殊的数据结构。如果你有一个应用程序要处理HTTP请求这种有点怪异的特性的话，请参见Django文档里关于Request和Response对象的章节。

标准类型小节

你会发现在这些标准类型里，列表和字典（“dict”）是应用程序里用的最频繁的数据结构。元组和字典主要是用在函数调用之间交换参数和返回值。字符串和数字也时有用到。Python还有更多数据类型可供选用，但是这里介绍的都是在编写Django应用过程中会经常碰到的类型。

1.5 流程控制

在了解Python变量的基础之后，我们现在来看看除了对它们赋值以外还能做点什么。如果不在它们之上加诸条件判断（根据条件选择代码执行的“路径”）和循环（根据某种形式的列表或者元组重复的执行一段代码）等逻辑，以变量形式出现的数据本身并没有太多的意义。

条件判断

Python和其他语言一样也提供了if和else语句。Python的“else-if”实际上是写成elif，这就和Bourne家族的脚本语言（sh、ksh和bash）是一样的。只要举一个简单的例子你就会明白它在Python里是怎么工作的了。

```
data = raw_input("Enter 'y' or 'n': ")
if data[0] == 'y':
    print "You typed 'y'."
elif data[0] == 'n':
    print "You typed 'n'."
```

```
else:
    print 'Invalid key entered!'
```

循环

和许多高级语言一样，Python也有while循环。while会重复执行同一段代码直到它的条件语句为假：

```
>>> i = 0
>>> while i < 5:
...     print i
...     i += 1
...
0
1
2
3
4
```

不过老实说，在Python里你不会经常需要这样使用while循环，因为我们有更加强大的循环机制for。在其他语言里，for只是作为一个可以计数的循环而存在，和它的搭档while差不太多。但是在Python这里，for则更像是shell脚本里的foreach循环，它能让你专注于问题本身而不用关心计数变量这种东西。

```
for line in open('/tmp/some_file.txt'):
    if 'error' in line:
        print line
```

如果你还记得列表推导式的话，你会发现其实这个例子就能转换成一个推导式。事实上很多简单的循环都可以（而且最好！）转换成列表推导式。不过有时候也需要简单的for循环，例如在调试的时候，你是没办法在列表推导式里使用print语句的。选择什么时候用列表推导式，什么时候用for循环是需要经验和时间的。

举个例子，enumerate是一个能让你同时迭代和计数的内置函数（for循环自身没办法计数）：

```
>>> data = (123, 'abc', 3.14)
>>> for i, value in enumerate(data):
...  print i, value
...
0 123
1 abc
2 3.14
```

> **在Django的Model中使用enumerate**
>
> 在编写Django代码时一个使用enumerate非常方便的地方就是model的定义，特别是在用到choices关键参数的地方——关于model field参数的细节请参见第4章。其用法如下：
>
> ```
> STATUS_CHOICES = enumerate(("solid", "squishy", "liquid"))
> ```

```
class IceCream(models.Model):
    flavor = models.CharField(max_length=50)
    status = models.IntegerField(choices=STATUS_CHOICES)
```

在数据库里，你的IceCream的status值被保存为整数（0,1,2），但是在Django的admin界面里显示的确实是文字标签。这样利用数据库存储的效率较高（如果你关心这个的话），而且当出现字符无法按照预想顺序进行排序的情况时也比较方便。

1.6 异常处理

和C++、Java等现代语言一样，Python也提供了异常处理。如同在本章开头中的例子，它想程序员提供了一种在运行时发现错误，进行恢复处理，然后继续执行的能力。Python try-except的结构和其他语言里的try-catch结构是很相似的。

如果在运行时发生异常的话，解释器会查找相应的处理语句（handler）。要是在当前函数里没有找到的话，它会将异常传递给上层的调用函数，看看那里能不能处理。如果在最外层（全局“main”）还是没有找到的话，解释器就会退出，同时打印出traceback以便让用户找出错误产生的原因。

记住虽然大多数错误都会导致异常，但一个异常不一定代表有错误发生。有时候它们只是一个警告，有时它们可以作为一个信号通知上层的调用函数，如退出循环等。

异常处理可以处理从简单的单个异常到一系列多个不同的异常类型。再用前面的那个例子，你可以看到这里我们处理了一个异常（except段里的代码）：

```
# attempt to open file, return on error
try:
    f = open(filename, 'r')
except IOError, e:
    return False, str(e)
```

同一段处理语句可以处理多个异常类型——只要把它们放到一个元组里就行了：

```
try:
     process_some_data()
except (TypeError, ValueError), e:
     print "ERROR: you provide invalid data", e
```

这个例子处理了两个异常，你可以在元组里放入更多异常。

当然你也可以为多个异常创建对应的多个处理语句：

```
try:
    process_some_data()
except (TypeError, ValueError), e:
    print "ERROR: you provide invalid data", e
except ArithmeticError, e:
     print "ERROR: some math error occurred", e
except Exception, e:
     print "ERROR: you provide invalid data", e
```

最后一个except语句利用了Exception是（几乎）所有异常的根类（root class）这一事实，所以如果有异常没有被它之前的任何语句捕捉的话，这个异常会在最后一个语句里被处理。

finally子句

Python还提供一个try-finally语句。我们不关心捕捉到的是什么错误，无论错误是不是发生，这些代码“必须”运行，比如关闭文件、释放锁，把数据库链接返还给连接池等。例如：

```
try:
    get_mutex()
    do_some_stuff()
finally:
    free_mutex()
```

当没有异常发生的时候，finally中的代码会在try完成之后立即运行。如果发生任何错误，那么finally的代码还是会被执行，但是它不会消除异常，异常依然还是会在调用链里寻找能处理它的语句。

自Python 2.5起，try-finally可以和except一起使用了。（之前的版本不支持这种写法。）

```
try:
    get_mutex()
    do_some_stuff()
except (IndexError, KeyError, AttributeError), e:
    log("ERORR: data retrieval accessing a non-existent element")
finally:
    free_mutex()
```

用raise抛出异常

到目前为止，我们只讨论了如何捕捉异常，那么怎么才能抛出异常呢？答案是使用raise语句。假设你在自己的库里创建了一个API调用要求传入一个大于0的正整数。在内置函数isinstance的帮助下（检验对象的类型），你代码看起来应该是这样的：

```
def foo(must_be_positive_int):
    """foo() -- take positive integer and process it"""

    # check if integer
    if not isinstance(must_be_positive_int, int):
        raise TypeError("ERROR foo(): must pass in an integer!")

    # check if positive
    if must_be_positive_int < 1:
        raise ValueError("ERROR foo(): integer must be greater than zero!")

    # normal processing here
```

表1.6列出了最常用和你在学习Python过程中很有可能会碰见的异常。

表1.6 常见的Python异常

异 常	描 述
AssertionError	assert（断言）语句失败
AttributeError	试图访问一个对象没有的属性，比如foo.x，但是foo没有属性x
IOError	输入/输出异常；基本上是无法打开文件
ImportError	无法引入模块或者包；基本上是路径问题
IndentationError	语法错误；代码没有正确对齐
IndexError	下标索引超出序列边界，比如当x只有三个元素，却试图访问x[5]
KeyError	试图访问字典里不存在的键
KeyboardInterrupt	Ctrl+C被按下
NameError	使用一个还未被赋予对象的变量
SyntaxError	Python代码非法，代码不能编译
TypeError	传入对象类型与要求的不符
UnboundLocalError	试图访问一个还未被设置的局部变量，基本上是由于另有一个同名的全局变量，导致你以为正在访问它
ValueError	传入一个调用者不期望的值，即使值的类型是正确的

当前异常的完整列表可以在官方文档 http://docs.python.org/lib/module-exceptions.html 里的exceptions模块一节中找到。

Django里的异常

和任何复杂的Python程序一样，Django也大量使用了异常。不过大多数情况下这些都是在内部处理，不属于日常使用Django的范围。但是有些异常却是需要你在Django应用里直接使用的。例如，抛出一个Http404的异常会通知Django转去处理HTTP 404“Not Found”的错误。这种在发生错误时能立即处理Http404错误的能力（而不是小心翼翼地通过传回一个特殊的标志）在创建Web应用里是非常好用的。

1.7 文件

在之前的代码里我们已经见过几个使用内置函数open的例子了。它的作用是打开文件以供读写：

```
>>> f = open('test.txt', 'w')
>>> f.write('foo\n')
>>> f.write('bar\n')
>>> f.close()
>>> f = open('test.txt', 'r')
>>> for line in f:
...     print line.rstrip()
...
foo
bar
>>> f.close()
```

除了write方法以外，还有一个read方法可以把整个文件的内容读入到一个字符串里。对文本文件来说，readlines方法能将文件里所有行读入到一个列表里，而writelines方法则可以将一个字符串列表用正确的断行符写回到文件里去。

一个文件对象本身就是一个迭代器，所以通常没有必要直接使用read或readlines 方法。一个简单的for循环在大多数情况下就已经足够了。

断行符（根据你的操作系统，可以是\n、\r\n、或\r）在读入行时是保留的，这就是为什么我们一定要调用字符串的rstrip方法来去掉它们。（否则输出会多出一个空行来，因为print会自动加上回车。）类似地，通过write或writelines方法发送给文件的字符串也需要断行符，否则它们会被合并为一行。

最后，另外这里没有论及的一些不太常用的文件方法都可以轻易地在《Core Python Programming》的“文件和I/O”一章里找到介绍。

1.8 函数

在Python里创建函数非常简单。在这一章里我们已经看过一些函数的声明了。这一节我们要讨论一些更深入的用法，包括（但不限于）：

- 声明和调用函数
- （函数调用里的）关键字参数
- （函数签名里的）默认参数
- 函数是first-class的对象
- 匿名函数和lambda
- （函数调用里的）参数容器
- （函数签名里的）变长参数
- 装饰器

声明和调用函数

声明函数的方法是用def关键字，然后是函数名以及小括号里的参数列表。（如果无参函数的话只要用一对括号即可）：

```
>>> def foo(x):
...     print x
...
>>> foo(123)
123
```

调用函数则更加简单：给出函数名和一对小括号，然后在括号里放入所需的参数就可以了。现在我们来看一点更实际的例子。

```
import httplib
def check_web_server(host, port, path):
    h = httplib.HTTPConnection(host, port)
    h.request('GET', path)
```

```
    resp = h.getresponse()
    print 'HTTP Response:'
    print '    status =', resp.status
    print '    reason =', resp.reason
    print 'HTTP Headers:'
    for hdr in resp.getheaders():
        print '    %s: %s' % hdr
```

这个函数的作用是什么？它接受一个主机名或者IP地址（host），服务器的端口号（port），以及一个路径名（path），然后尝试和运行在指定主机和端口号上的Web服务器连接。如果成功的话，它会向给定的路径提交一个GET请求。若执行下面的这段代码检查Python网站的服务器，你会得到的输出如下：

```
>>> check_web_server('www.python.org', 80, '/')
HTTP Response:
    status = 200
    reason = OK
HTTP Headers:
    content-length: 16793
    accept-ranges: bytes
    server: Apache/2.2.3 (Debian) DAV/2 SVN/1.4.2 mod_ssl/2.2.3 OpenSSL/0.9.8c
    last-modified: Sun, 27 Apr 2008 00:28:02 GMT
    etag: "6008a-4199-df35c880"
    date: Sun, 27 Apr 2008 08:51:34 GMT
    content-type: text/html
```

（函数调用里的）关键字参数

除了这样“常规的”调用约定，Python还允许你指定关键字参数（keyword argument），这可以让函数更加清晰，更加容易使用，同时，也消除了记忆固定参数顺序的需要。关键字参数通过“键=值”这样的形式加以指定，下面是对之前例子里的一点修改（输出被省略了）：

```
>>> check_web_server(port=80, path='/', host='www.python.org')
```

根据给出的关键字，Python就会在执行程序时将相应的对象正确的赋值给那些变量。

（函数签名里的）默认参数

Python函数的另一个特点是可以为参数提供默认值，这样向它们传递参数就不是必需的了。很多函数都有一些对所有调用值都一样的变量，在这种情况下定义默认值就可以让函数好用一点。

参数的默认值是在函数签名里用等于号直接指定的。拿刚才检查网站的代码为例，大多数网站都是运行在80端口上，并且检查“网站是否运行”通常只要测试首页就行了。所以我们可以加入这样的默认值：

```
def check_web_server(host, port=80, path='/'):
```

这里，不要和关键字参数弄混淆了。关键字参数只能用于“函数调用”，而默认参数则是用于“函数声明”。所有必须提供的参数一定要出现在任何可选参数之前，不能混在一起或者颠倒顺序。

```
def check_web_server(host, port=80, path):          # INVALID
```

将列表和字典作为默认参数

这里我们要提醒你注意一个Python初学者常犯的错误。还记得之前我们讨论的关于可变和不可变的变量么？列表和字典是可变的，而字符串和整数是不可变的。因为这样，将列表和字典作为默认参数可能会非常危险，特别是当它们持续地穿过多个函数调用时，像这样：

```
>>> def func(arg=[]):
...     arg.append(1)
...     print arg
...
>>> func()
[1]
>>> func()
[1, 1]
>>> func()
[1, 1, 1]
```

这种可变对象的特性不是非常明显，这也是为什么我们要在这里特别提到的原因。一不小心你就会发现有些函数会引入非常奇怪的行为。

函数是First-Class对象

在Python里你可以把函数（和方法）当作和其他对象一样的使用，例如，把它们存放在容器里，赋值给其他变量，作为参数传递给函数等。唯一的不同是你可以这个函数对象，就是说当附上括号和参数时，你就把它当作了一个函数来对待。在深入讨论之前，我们先要理解对象引用的概念。

引用

当执行def语句的时候，你实际上是在创建一个函数对象并将其赋值或绑定到当前名字空间里的一个名字上，但这可以只是它许多引用（或别名alias）里的第一个。每次当你向另一个函数调用传递一个函数对象，将其放入一个容器，或是赋值给一个局部变量时，你就为那个对象创建了一个额外的引用，或者说别名。

作为一个例子，下面的代码在全局名字空间里创建了不是一个，而是两个变量，因为函数一经定义，它就和其他变量没什么两样了：

```
>>> foo = 42
>>> def bar():
...     print "bar"
...
```

和其他Python对象一样，函数可以有任意多的引用。下面是一些例子来使之更容易理解。首先是bar的正常用法：

```
>>> bar()
bar
```

接着，我们将bar赋值给另一个名字baz，如此之前的函数bar现在也可以通过baz来引用了。

```
>>> baz = bar
>>> bar()
bar
>>> baz()
bar
```

要使用一个存放在容器里的函数对象，你只需和其他对象一样引用它，然后跟上小括号并且传入参数就可以了。例如：

```
>>> function_list = [bar, baz]
>>> for function in function_list:
...     function()
...
bar
bar
```

注意只有当我们想要调用函数的时候才会加上小括号。在把它当作变量或者对象传来传去的时候，你只需要用函数的名字（就像上面我们创建function_list时做的那样）就行了。这将引用函数对象或者对象的名字（比如bar）和实际调用或执行它（比如bar()）区别开来了。

Django里的First-Class函数

Django充分利用了Python函数对象可以像其他值一样被传递这一特性。最常见的例子就是在URLconf文件里设置Django view。

```
from django.conf.urls.defaults import *
from myproject.myapp.views import listview

urlpatterns = patterns('',
    url(r'^list/', listview),
)
```

在这段代码里，listview作为一个函数对象被直接使用，而不是一个包含了函数名的字符串。

另一个利用函数对象的地方是在model field里的默认参数。例如，如果你希望DateField在默认情况下就能收到创建时间，你可以在它被调用时传入一个标准库的函数对象。

```
import datetime


class DiaryEntry(models.Model):
    entry = models.TextField()
    date = models.DateField(default=datetime.date.today)
```

这有点不太好理解。如果我们将default设置为datetime.date.today()的话——注意这里的括号——函数将会在model定义的时候被调用，这不是我们想要的，所以我们传递的是函数对象。Django会意识到这一点，并且在创建实例的时候调用函数，为我们生成需要的值。

匿名函数

匿名函数是Python提供的另一个函数编程的特性。创建的方法是使用lambda关键字，它由一个表达式组成，这个表达式代表了函数的“返回值”。这样的函数和普通函数声明的方式不同，故而没有名字，所以也称为匿名函数。它们通常只是一行表达式，所以一般都是用完就扔。这里我们要弄清楚“表达式”expression和“语句”statement之间的区别以避免任何混淆。

表达式 v.s. 语句

Python代码由表达式和语句组成，并由Python解释器负责执行。它们的主要区别是“表达式”是一个值，它的结果一定是一个Python对象。当Python解释器计算它时，其结果可以是任何对象。例如42、1 + 2、int('123')、range(10)等。

结果不是对象的代码则称为“语句”。例如if或者print语句，for和while循环等。它们执行一个动作，而不是返回或生成一个值。

使用lambda

lambda的语法如下：lambda args:表达式。在执行的时候，lambda返回一个可以立即使用的函数对象，或者你可以选择将它保存为一个变量，或是保存为一个回调函数以便稍后执行。

lambda的一个常见用法就是为sorted这样的函数工具提供一个函数对象，它在众多参数里有一个key参数，这个key必须是一个函数，可以用在列表里的每个要排序的元素上以生成一个值来作为排序的依据。例如有一个代表人物的复杂对象列表，我们希望按照他们的姓氏属性来排序的话，则可以这样：

```
sorted(list_of_people, key=lambda person: person.last_name)
```

这是正确的原因是由于key期望的是一个函数对象，而lambda返回的正好是一个匿名函数。当然下面这样的写法也是等价的：

```
def get_last_name(person):
    return person.last_name

sorted(list_of_people, key=get_last_name)
```

甚至我们还可以这样写：

```
get_last_name = lambda person: person.last_name
sorted(list_of_people, key=get_last_name)
```

这三个语句最大的不同之处在于可读性和重用性。第一个例子更简洁同时目的相当明确，通常这是解决这类问题最好的手法。不过，也有很多lambda最终“成长”为一个普通的函数（例如当程序员意识当他要重复使用这个函数时），这时，第二个例子就比较恰当。

第三个例子其实不太现实（我们希望lambda是一种一次性的函数定义），不过它也进一步地展示了Python函数First-Class的特性。

> **Django里的lambda函数**
>
> Django里用到lambda函数的地方不太多，但是有一个地方看起来特别的适合：即所谓的“认证装饰器”（authentication decorator），它的作用是确认用户有足够的权限访问某些

页面。一种办法是将一个代表已登录用户的User对象传递给一个函数，如果允许用户访问就返回True，反之则返回False。

这样的函数可以用一般的def foo(): 结构来定义，但是lambda的形式更加简洁。你现在还不具备理解这个例子的全部知识点，但是这不影响我们理解它的作用。

```
@user_passes_test(lambda u: u.is_allowed_to_vote)
def vote(request):
    """Process a user's vote"""
```

以@开头的行就是一个“函数装饰器”(function decorator)，在本章稍后你会学到这个概念。装饰器通过“包装”函数（例如这里的vote函数）来改变它们的行为。这里的user_pass_test装饰器是Django内置的一个特性，它接受一个任何接受Django User对象的函数作为参数，并返回一个布尔值（True或False）。因为这里的测试非常简单（我们只是简单地返回了User对象里一个特定的属性），只需一行代码就可以了。

*args和**kwargs

这一节我们要讨论Python里*和**的特殊含义，它们都和函数有关但是在函数调用时和函数声明时却有着不同行为。在开始之前，我们先要对所有C/C++程序员讲明，这里的星号和指针可没有任何关系！

一般说来，无论是函数调用还是声明，单个星号表示有元组（或是列表）“出现”，而两个星号则代表附近有字典“出没”。我们先从函数调用开始。

函数调用里的*和**

我们来重新看一下之前的例子check_web_server函数。下面是函数的签名：

```
def check_web_server(host, port, path):
```

用check_web_server('127.0.0.1', 8000, '/admin/')即可调用这个函数。那要是这些信息在一个三元组里的话怎么办？例如：

```
host_info = ('www.python.org', 80, '/')     # http://www.python.org/
```

这时调用就会变成：

```
check_web_server(host_info[0], host_info[1], host_info[2])
```

然而这种写法既不可扩展（要是函数有超过一打的参数怎么办？）也不好看。这时单星号就可以解决我们的问题，因为当调用函数时，表达式在计算一个带星号前缀的元组或列表时会将其“打开”。所以下面的例子和上面的代码是等价的：

```
check_web_server(*host_info)
```

这样的写法就明显要干净和优雅得多，而双星号对字典的用法也是类似的。现在我们来创建一个具有和('www.python.org', 80, '/')相似内容的字典。

```
host_info = {'host': 'www.python.org', 'port': 80, 'path': '/'}
```

于是函数的调用就变成了：

```
check_web_server(**host_info)
```

这告诉函数在打开字典时，每个键是参数的名字，同时对应的值是函数调用的参数。即，它就等于：

```
check_web_server(host='www.python.org', port=80, path='/')
```

你可以同时使用这些技术，这和手动的通过下标或是关键字参数调用函数是一样的。

*函数签名里的*和***

函数签名里的*和**虽然看着相似但作用却完全不同：它们让Python得以支持变长参数，有时也称为“varargs”。即函数可以接受任何数量的参数。

当定义一个有三个参数的函数时（没有默认值的参数），调用者必须传入正好三个参数。默认参数虽然引入了一些灵活性，但函数依然受制于所定义参数的最大数目。

如果需要更大的灵活性，则可以用单星号表示的元组来定义一个变长参数，就好像传入一个“购物袋”一样包含了所有元素。现在我们来创建一个这样的“daily sales total”函数：

```
def daily_sales_total(*all_sales):
    total = 0.0
    for each_sale in all_sales:
        total += float(each_sale)
    return total
```

相应的合法的函数调用有：

```
daily_sales_total()
daily_sales_total(10.00)
daily_sales_total(5.00, 1.50, '128.75') # Any type is allowed, not just floats!
```

不管你向这个函数传递多少参数，它都能够处理。all_sales就是一个包含了所有参数的元组（这也是为什么我们可以在函数定义里迭代all_sales的原因）。

你还可以把普通参数和变长参数混在一起使用，这时vararg就会捕捉所有剩下的参数，例如现在这个假设的check_web_server函数定义就能接受额外的参数了。

```
def check_web_server(host, port, path, *args):
```

> **注意**
>
> 在函数定义里使用变长参数时，所有必须出现的参数一定先出现，然后是有默认值的参数，最后才是变长参数。

类似地，你也可以在函数签名里用双星号来接受任意数目的关键字参数，它们会在函数被调用的时候导入到一个字典里去。

```
def check_web_server(host, port, path, *args, **kwargs):
```

如此，这个函数必须至少接受初始三个参数，但是也能接受随后任何数目的参数或是关键字参数；在函数内部，我们可以分别检查args元组和kwargs字典的内容来决定是否丢弃它们。

实际上，还有一种全部由变长参数组成的所谓“通用”Python方法签名（“universal” Python method signature）。

```
def f(*args, **kwargs):
```

这样的函数可以按照f()，f(a, b, c)，f(a, b, foo=c, bar=d)等任何方式调用，它可以接受任何形式的输入。当然了，函数内部如何处理args以及kwargs的内容则取决于它的作用。

Django QuerySets里的kwargs：动态创建ORM查询**

Django的数据库API查询经常包含关键字参数。例如：

```
bob_stories = Story.objects.filter(title__contains="bob",
subtitle__contains="bob", text__contains="bob",
byline__contains="bob")
```

够简单吧。下面的例子展示了如何按字典的形式传递这些关键字参数：

```
bobargs = {'title__contains': 'bob', 'subtitle__contains': 'bob',
    'text__contains': 'bob', 'byline__contains': 'bob'}
bob_stories = Story.objects.filter(**bobargs)
```

这样，你就可以动态地创建字典了：

```
bobargs = dict((f + '__contains', 'bob') for f in ('title', 'subtitle', 'text',
    'byline'))
bob_stories = Story.objects.filter(**bobargs)
```

这样，你就可以用这些技术来精简流水化查询（streamlining query）中的冗余，甚至有助于组装动态获取的过滤参数（例如从搜索表单里得到的选项）。

装饰器

学习Python函数和函数式编程中最后一个可能也是最难懂的概念就是装饰器（decorator）。在这里，Python的装饰器指的是一种让你能改变或者说“装饰”函数行为的机制，它能让函数执行一些和原本设计不同，或是在原有基础上额外的操作。装饰器是也可以说是对函数的一个包装。这些额外的任务包括写日志、计时、过滤等。

Python里一个被包裹或被装饰的函数（对象）通常会被重新赋值给原来的名字，这样被包裹的函数能和普通的版本保持兼容——因为使用装饰器就是在现有的功能上再“加盖”额外的功能。

最简单的语法形式是这样的：

```
@deco
def foo():
    pass
```

在这个例子里，deco就是一个“装饰”了foo函数的装饰器函数（decorator function）。它先把foo函数拿过来，加上一些额外功能后再重新赋值给foo。@deco的语法和下面的代码是等价的（假设这里的foo是一个有效的函数对象）：

```
foo = deco(foo)
```

下面这个简单的例子记录了函数调用的发生：

```
def log(func):
    def wrappedFunc():
        print "*** %s() called" % func.__name__
        return func()
    return wrappedFunc

@log
def foo():
    print "inside foo()"
```

现在我们来看看执行代码后所生成的结果：

```
>>> foo()
*** foo() called
inside foo()
```

在本章稍早的地方，我们看见过这样一个接受一个参数的装饰器。

```
@user_passes_test(lambda u: u.is_allowed_to_vote)
```

在这个例子里，实际上是调用了一个函数，然后返回一个事实上的装饰器——user_passes_test自己并不是一个装饰器，它只不过是一个接受了参数的函数，然后用这些参数返回一个可用的装饰器。它的语法是这样的：

```
@decomaker(deco_args)
def foo():
    pass
```

这个下面的代码是等价的，注意这里Python表达式是如何串联在一起的：

```
foo = decomaker(deco_args)(foo)
```

这里的“装饰器生成器”（decorator-maker，或者叫decomaker）接受参数deco_args并且返回一个可以包装foo函数的装饰器。

最后一个例子展示了如何同时应用多个装饰器：

```
@deco1(deco_args)
@deco2
def foo():
    pass
```

这里我们不再多做深入讨论了，不过根据现有的代码，你可以认定它和下面的代码是等价的：

```
foo = deco1(deco_args)(deco2(foo))
```

你或许会想，“为什么要用装饰器呢”？实际上，包裹函数的概念在Python里不算什么新的东西，同样接受对象，修改并重新赋值给原来的变量也不是新的概念。不同的是装饰器可以让你用一个简单的@符号完成这一切动作。

如果你需要一份更完整更易读的装饰器教程，请访问Kent John的“Python Decorators”教

程，http://personalpages.tds.net/~kent37/kk/00001.html。

1.9 面向对象编程

首先声明，这一节不是一份面向对象编程（OOP）的教程。我们只不过是准备介绍一下如何在Python里创建和使用类。如果你不知道什么是OOP的话，使用Python是学习它最简单的方法之一——当然最好你还是先读一下其他高级教程，不过我们不做强求。OOP的主要目标是在代码和现实问题之间提供一个合乎逻辑的映射关系，并且鼓励代码的重用和共享。我们还会展示一下Python独有的特性。

类的定义

我们选用的第一个建模现实世界里的问题是创建一份通信录。要在Python里创建一个类，你需要提供class关键字，这个新的类的名字，以及一个或多个基类（base class）。例如，如果要创建一个Car（轿车）或是Truck（卡车）类的话，你可以用Vehicle（车辆）作为一个基类。如果你不想从任何现有类继承的话，只需使用Python的根（root）类或类型object作为你的基类就行了，就像这里我们创建的“通信录条目”类，AddressBookEntry：

```
class AddressBookEntry(object):
    version = 0.1

    def __init__(self, name, phone):
        self.name = name
        self.phone = phone

    def update_phone(self, phone):
        self.phone = phone
```

version这样类的静态成员（Static class member）的创建是通过在类定义里赋值完成的。这样的成员就是属于整个类的变量，可以在所有实例之间共享。方法（method）的定义和函数一样，就是每个方法都必须带上一个额外的self对象——Python在这一点上非常明确。

变量self指向的是类的一个特定的实例（在其他语言里用this来代表这个self的概念）。如果说类是一份图纸，那么实例就是它的一份具体实现，一个在执行过程中创建出来让你操作的真实对象。所有以self开头以及点-属性（dotted-attribute）形式出现的变量都是一个实例属性，即一个属于特定实例的对象。如果name是一个实例属性的话，你必须用self.name作为完整的名字来引用它。

如果你有使用其他有面对对象特性语言的经验的话，请注意Python没有构造函数和析构函数的概念——也没有new和free关键字。稍后我们会更详细地讨论这一点。

实例化

其他语言一般都使用new语句来创建实例，不过在Python里你只要像调用函数一样调用类的名字就可以了。Python用的是“初始化程序”（initializer）而不是“构造函数”（constructor），

所以名字用的也是__init__。在实例化一个对象时，你需要传入__init__所需的参数。在Python在创建对象时会用你提供的参数自动调用__init__，并最终返回将一个新创建的对象返回给你。

我们说的是调用方法，不过其实也完全可以说是函数调用。虽然你在方法声明里被要求提供self参数，但是在（以常规的方式）调用方法时却不用这么麻烦，Python会自动为你传入self对象。下面的例子创建了两个AddressBookEntry类的实例：

```
john = AddressBookEntry('John Doe', '408-555-1212')
jane = AddressBookEntry('Jane Doe', '650-555-1212')
```

Python会为每个实例调用__init__，随后返回这个对象。仔细看，这里并没有在调用时给出self，只有name和phone number。Python会为你传入self的。

现在你就可以直接访问属性了，比如john.name、john.phone、jane.name和jane.phone。如同下面的例子，基本上对实例属性的访问没有什么限制：

```
>>> john = AddressBookEntry('John', '408-555-1212')
>>> john.phone
'408-555-1212'
>>> john.update_phone('510-555-1212')
>>> john.phone
'510-555-1212'
```

再次强调一下，注意这里的update_phone方法实际上有两个参数，self和newphone。但是我们只需要提供新的phone number就可以了，Python会自动将指向john的实例对象作为self传递进来。

Python还支持动态的实例属性，即那些没有在类定义里声明的属性，可以“凭空”创造出来。

```
>>> john.tattoo = 'Mom'
```

这是一个非常好用的特性，完全展示了Python的灵活性。你可以随时随地创建任意数目的属性。

继承

创建一个子类和创建一个类是很相似的，你只需要提供一个或多个基类（不是对象）就行了。继续刚才的例子，现在我们来创建一个员工通信录条目类。

```
class EmployeeAddressBookEntry(AddressBookEntry):
    def __init__(self, name, phone, id, social):
        AddressBookEntry.__init__(self, name, phone)
        self.empid = id
        self.ssn = social
```

注意这里我们没有给self.name和self.phone分配对应的name和phone number——这是因为在AddressBookEntry.__init__调用里已经为我们处理了。当你按这种方式重写某个基类方法的时候，你必须显式地调用它（原来的方法），就像上面这个例子那样。注意这一次我们必须传入self参数，因为我们指向的是AddressBookEntry类而非实例。

除了那些被重写的基本基类方法外，子类还会继承所有一切，所以这里EmployeeAddressBookEntry也同样拥有name和phone属性，以及update_phone方法。和绝大多数Python程序和框架一样，Django在其内部以及用户关心的外部特征里也都大量使用了继承。

嵌套类

就像用来创建装饰器的“嵌套函数”（inner function）一样，你还可以创建嵌套类（inner class），即在类的内部定义的类，例如：

```
class MyClass(object):
    class InnerClass:
        pass
```

嵌套类也是一个真正的Python类，但它只有在MyClass实例的内部才可见。这是一个Python中有点生僻的特性，不过在Django里用到的地方却很多（请参见下面的边栏，“类和Django Model”）。你在Django里唯一一个可能用到的嵌套类（但是是非常重要的一个）就是Meta嵌套类。

类和Django Model

Django的数据model（绝大多数Django应用的核心）是从Django的内置类django.db.models.zModel继承而来的子类。Django的Model类提供了大量的强大特性，我们将在第4章里做深入介绍。现在，我们先用第2章里的一段代码来做例子：

```
from django.db import models
from django.contrib import admin

class BlogPost(models.Model):
    title = models.CharField(max_length=150)
body = models.TextField()
timestamp = models.DateTimeField()

class Meta:
    ordering = ('-timestamp',)
```

这里定义了一个从django.db.models.Model继承而来的新类BlogPost，它有三个自定义的变量。继承Model还给予了BlogPost其他方法和属性，包括了允许你从数据库里查询BlogPost对象，创建新的对象，以及访问相关的条目。

1.10 正则表达式

Django使用了一种叫做正则表达式（regular expression）的字符串匹配机制来定义网站的URL。如果没有正则表达式（也常称为“regex”）的话，我们将不得不定义每一个可能的URL，在对 /index/ 或 /blog/posts/new/的时候可能还好，但是遇上稍微动态一点的URL，像/blog/posts/2008/05/21/ 这样的URL就很麻烦了。

很多书籍和在线教程都介绍了究竟什么是正则表达式以及如何编写它们，所以这里我们就不花太多时间在这个话题上了。你可以到withdjango.com的网站上找到一些很好的资源。本节的余下部分会假设你已经熟悉了regex的使用（至少也是读过一点教程了），现在我们来看看它们在Python里的用法。

re模块

Python的正则表达式可以通过re模块来访问，这是在查找函数中使用非常频繁的一个组件。re.search返回一个匹配对象，随后可以用这个对象的group或groups方法获取匹配的模式。

```
>>> import re
>>> m = re.search(r'foo', 'seafood')
>>> print m
<_sre.SRE_Match object at ...>
>>> m.group()
'foo'
>>> m = re.search(r'bar', 'seafood')
>>> print m
None
```

search函数会在成功时返回一个Match对象，调用它的group方法就可以得到匹配的字符串。当它失败的时候，你得到的是None。注意这里用到的raw字符串标志r"——和本章之前提到的一样，正则表达式最好用raw字符串来表达，这样就能避免大量使用反斜杠这样的转义符。

查找和匹配

这里我们要区分“查找”和“匹配”的概念。查找是在目标字符串里搜寻任何匹配的模式，而匹配则表示整个字符串都必须要符合模式的描述。例如：

```
>>> import re
>>> m = re.match(r'foo', 'seafood')
>>> if m is not None: print m.group()
...
>>> print m
None
>>> m = re.search(r'foo', 'seafood')
>>> if m is not None: print m.group()
...
'foo'
```

因为r'foo'只能匹配'seafood'的一部分，所以调用re.match的结果为空（None）。但是re.search则稍微宽容一点，所以我们能得到一个非空的结果。

1.11 常见错误

我们在这一节里要讨论的是一些Python新手经常容易犯的错误，比如：怎么做才能创建一个只有一个元素的元组？又或者，为什么在面向对象的Python代码里到处都是self?

单元素的元组

初学者虽然能明白()和(123, 'xyz', 3.14)都是元组，但是却不太能意识到(1)其实不是一个元组。Python实际上是在多处重载了小括号。比如当用在表达式里的时候，小括号的作用是分组。如果你想要一个单元素的元组，在那个元素后面必须跟一个不太漂亮不过一定要有的尾巴（逗号）好像这样(1,)。

模块

我们已经见过很多种导入模块和它们属性的方法了：

```
import random
print random.choice(range(10))
```

和

```
from random import choice
print choice(range(10))
```

第一种方法是将模块的名字设置为一个隐含的名字空间里的全局变量，这样你就可以好像访问全局属性那样访问choice函数。而在第二个例子里，我们是直接把choice引入到全局名字空间里来（而非模块的名字）。因此不再需要把这个属性当成是模块的成员了。实际上我们也只拥有了这个属性而已。

Python新手之间经常有一种误解，以为第二种方法只导入了一个函数，而没有导入整个模块。这是不对的。整个模块其实已经被导入了，但是只有那个函数的引用被保存了起来。所以from-import这种语法并不能带来性能上的差异，也没有节省什么内存。

能不能重复导入一个模块

新手经常会担忧的一个问题是他们有两个模块m.py和n.py都导入了foo.py模块。当m导入n时，foo岂不是会被导入两次？简单的来说，没错，是这样的，但是和你想的有点不一样。

Python有导入模块（importing）和加载模块（loading）之分。一个模块可以被导入任意多次，但是它只会被加载一次。就是说，当Python碰到一个已经被加载的模块又被导入时，它会跳过加载的过程，所以你无需担心额外消耗内存的问题。

Package

Package是Python在文件系统上发布一组模块的一种方式，它使用常见的dotted-attribute方式来访问子模块，就好像它们是另一个对象的属性而已一样。或者说，如果你发布的软件产品里有几百个模块的话，基本上是不可能把它们都放在同一个目录下的。当然这样不是不行，但你真的想这么做么？为什么不利用文件系统来用一种更符合逻辑更直观的方式组织模块呢？

例如。假设有一个电话程序。我们可以组织这样的一个目录结构：

```
Phone/
    __init__.py
    util.py
    Voicedata/
```

```
        __init__.py
        Pots.py
        Isdn.py
    Fax/
        __init__.py
        G3.py
    Mobile/
        __init__.py
        Analog.py
        Digital.py
```

Phone是顶层目录，或者说package。在它之下有子package，不过实际上它们就是包含了其他Python模块的子目录罢了。你可以看到每个子目录下都有一个_init_.py文件。这告诉Python解释器这些目录下的文件应该被当作是一个子package而不是普通文件。它们一般都是空文件，当然也可以在使用任何子package代码之前做一些初始化的工作。

假设现在我们要访问analog cell phone的dial函数。我们可以执行：

```
import Phone.Mobile.Analog
Phone.Mobile.Analog.dial()
```

这看着有点笨拙。在实际工作中，基于效率考量你可能会抄一点近路，当然更多原因自然是为了可以少打几个字母啦！Python和Django都是以简单易用著称，尽量遵循DRY的原则。下面的例子可能更加贴近现实一点：

```
import Phone.Mobile.Analog as pma
pma.dial()
```

可改变性

初学者经常会问的一个问题是，到底Python是“传引用的”还是“传值的”？这个问题的答案没办法用简单的是与不是来回答，所以只能说是“看情况”——有的对象在传入函数时是一个引用，而有些则是被复制进来，即传值。而判断的依据就是看对象的可改变性（mutability），而这一点又取决于对象的类型。由于这种双重行为，Python程序员通常不用“传引用”或是“传值”这种说法，取而代之的是对象是可变的（mutable）还是不可变的（immutable）。

可改变性指的是一个对象的值能否改变。所有Python对象都有三个属性：类型、标识符，和值。类型和值的意思很明显，标识符指的是运行在解释器里所有对象都有的唯一的一个标识号。

所有三个属性几乎总是只读的，即在对象的生命期间它们是不可改变的。唯一的例外就是对象的值：如果这个值可以改变，那它就是一个可变的对象，反之则是不可变的对象。

简单类型或者“标量”（scalar）类型，包括整数等其他数字类型，str和unicode这样的字符串类型，以及元组都是不可变的。剩下的如列表，字典，类，类实例等都是可变的。

> **注意**
>
> 在这一节里你会了解到，可改变性在Python编程里是非常重要的一个部分。这也解释

了为什么Python要提供两种“列表”类型，列表list和元组tuple，其中列表是可变的而元组不是。这种能够提供“仿列表”对象的能力在需要不可变性的场合下（例如字典里所有键）会变得非常有用。

可改变性如何影响方法调用

调用可变对象的方法是一个需要小心谨慎对待的区域。如果你调用的函数有任何修改一个可变对象的行为的话，通常它就是直接修改的，即直接修改其数据结构而不是返回一个修改后对象的拷贝（即这类函数返回的都是None）。

一个常用的例子是list.sort，它就是直接在列表上排序而不是返回它，这很容易把很多Python新手彻底搞晕！很多其他列表方法，如reverse和extend，以及字典方法update（向字典添加新的键值对）等也都是直接就地修改对象。

好在Python从2.4版起提供了一些sorted和reversed这样的内置函数，它们接受一个可迭代的对象作为输入，并且返回一个排好序或者倒置的拷贝。这在不希望修改原对象或者是希望节省几行代码的情况下非常有用。如果你一定要用Python 2.3的话，你需要手动复制这个列表（一般是通过newlist = list(mylist) 或 newlist = mylist[:]），随后在新的拷贝之上调用sort方法来获取一份修改过的拷贝。

复制对象和可改变性

现在我们来看一个Python新手很容易犯的错误，可改变性和复制对象。在本节开始的时候，我们提到了不可变对象是传值的，而可变对象是传引用的。不管是向函数传递参数或是任何形式的对象复制来说，这都是一样的：不可变对象（比如整数）被真正复制，而可变对象只是复制了一个对它们的引用，即在内存中只有一份对象，而有两份引用。

为什么这一点如此重要呢？请看下面这个嵌套列表的例子：

```
>>> mylist = [1, 'a', ['foo', 'bar']]
>>> mylist2 = list(mylist)
>>> mylist2[0] = 2
>>> mylist[2][0] = 'biz'
```

你在这里希望对嵌套列表mylist的修改不会影响到mylist2，但实际上不是这样的！看看这两个列表的新值。

```
>>> print mylist
[1, 'a', ['biz', 'bar']]
>>> print mylist2
[2, 'a', ['biz', 'bar']]
```

mylist中前两个对象都是不可变的（一个是整数和一个字符串），所以mylist2得到了两个全新的整数和字符串对象，所以把1替换成2这一步完全正确。但是，mylist的第三个对象是一个列表，而列表是可变的，所以mylist2得到的只是一份它的引用。因此两个列表中的第三个对象其实都只是一份指向内存中列表对象的引用，在任何一个父列表中修改它都会影响到另一个。

这种类型的复制称之为“浅拷贝”（shallow copying），因为它只复制了对可变对象的引用而没有尝试获取对象的值。如果确实需要后面这种行为，即“深拷贝”（deep copying），你必

须导入copy模块中的copy.deepcopy。请在使用前仔细阅读相关文档——这种类型的拷贝通常都是递归的（如果有循环引用的话就会产生问题），而且不是所有对象都是可以深拷贝的。

构造函数v.s. 初始化程序

虽然Python是面向对象的语言，但是它和传统OOP语言的一个区别之处在于它没有显式的构造函数的概念。Python里没有new关键字因为你并没有真的实例化你的类。相反，Python会为你创建实例并调用初始化程序——这是在你的对象被创造出来之后，但是在Python将它返回给你之前调用的第一个方法。它的名字是__init__。

要实例化一个类，或者说要创建一个对象，你可以像调用函数一样调用这个类。

```
>>> class MyClass(object):
...     pass
...
>>> m = MyClass()
```

此外，由于Python会为你自动调用__init__，如果它需要参数的话，你必须在“调用”类的时候提供给它。

```
>>> from time import ctime
>>> class MyClass(object):
...     def __init__(self, date):
...         print "instance created at:", date
...
>>> m = MyClass(ctime())
instance created at: Wed Aug  1 00:59:14 2007
```

类似的，Python程序员通常不需要实现析构函数来销毁对象，只需让它离开作用域就行了（这时它们会被自动垃圾回收）。不过，你可以在Python对象里定义一个__del__方法来当作析构函数，而且你可以用del语句来显式地销毁一个对象（例如，del my_object）。

动态实例属性

Python新手另一个可能会搞混的东西（特别是来自其他面向对象语言背景的程序员）是实例的属性可以动态分配，即使是在类定义已经完成甚至已经创建实例以后。例如这里的AddressBook类：

```
>>> class AddressBook(object):
...     def __init__(self, name, phone):
...         self.name = name
...         self.phone = phone
...
>>> john = AddressBook('John Doe', '415-555-1212')
>>> jane = AddressBook('Jane Doe', '408-555-1212')
>>> print john.name
John Doe
>>> john.tattoo = 'Mom'
>>> print john.tattoo
Mom
```

注意这里的self.tattoo没有在类、方法声明和类方法的任何地方出现过——甚至在__init__里也没有！我们能够在运行时动态的创建属性。这就是动态实例属性（dynamic instance attribute）。

1.12 代码风格

Python里有很多“正确的编码风格”的元素、推荐和建议，但是总的来说就是要保持代码的味道要够“Pythonic”。遵循Python保持简洁，DRY（不要重复自己），易读的代码，鼓励优雅的方案和代码重用等设计哲学的编程系统都算是Pythonic的标签，Django也不例外。在这里有限的篇幅里，我们只能提供一些基本的指导。剩下的部分则来自Python、Django的经验，以及它们高度活跃的社区。Python有一份官方的风格指导，你可以在PEP8里找到它(http://www.python.org/dev/peps/pep-0008)。

四空格对齐

在一个用空格区分代码块的语言里，再加上各种各样的用户类型，如果只用一个或是两个空格来对齐Python代码的话，编辑起来就实在是太伤眼睛了。同样用八空格的话，代码又太容易折行。Guido在最早的文章中就一直建议使用四个空格是一个完美的折中。

使用空格而非Tab

无论在什么平台上开发，你的代码总是有可能会被移动或是复制到另一个不同架构的机器上，或是运行在一个不同的操作系统上。由于制表符（tab）在不同平台上处理的方式都不同（例如在Win32上是四个空格，而到了POSIX或UNIX类系统上就变成了八个），所以最好还是干脆就避免使用tab。

如果Python解释器向你报错说代码有错误，但是从屏幕上看起来一切正常的话，很有可能就是代码的什么地方混进了tab因此编辑器“错误地”显示了你的代码。在显式地把所有tab都转换成空格后，你就可以发现哪里的对齐有问题了。

不要像标题一样把一组代码写在同一行里

虽然这样的代码是完全合法的：

```
if is_finished(): return
```

但是我们还是推荐你把它分开写到多行里去，像这样：

```
if is_finished():
    return
```

这里的主要原因是可读性，另外当你要往里添加额外代码时无需更多编辑。

创建文档字符串（即“docstring”）

代码的文档是非常有用的，Python不仅允许你为代码创建文档，还让你可以在运行时访问它。模块、类，以及函数和方法都可以创建docstring。比如下面这个简单的例子foo.py：

```
foo.py:
#!/usr/bin/env python
"""foo.py -- sample module demonstrating documentation strings"""

class Foo(object):
    """Foo() - empty class ... to be developed"""

def bar(x):
    """bar(x) - function docstring for bar, prints out its arg 'x'"""
    print x
```

这里说的“可以在运行时访问”指的是：如果你启动解释器并导入模块时，你可以通过每个模块、类和函数的__doc__属性来访问它的docstring。

```
>>> import foo
>>> foo.__doc__
'foo.py -- sample module demonstrating documentation strings'
>>> foo.Foo.__doc__
'Foo() - empty class ... to be developed'
>>> foo.bar.__doc__
'bar(x) - function docstring for bar, prints out its arg "x"'
```

此外，你还可以用内置函数help来“漂亮地”打印docstring。这里我们只给出一个模块的例子（你可以自行实验类和函数的docstring）。

```
>>> help(foo)
Help on module foo:
NAME
    foo - foo.py -- sample module demonstrating documentation strings

FILE
    c:\python25\lib\site-packages\foo.py

CLASSES
    __builtin__.object
        Foo

    class Foo(__builtin__.object)
     |  Foo() - empty class ... to be developed
     |
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)

FUNCTIONS
    bar(x)
        bar(x) - function docstring for bar, prints out its arg "x"
```

Python标准库里的大多数软件都具有docstring，所以要是你需要内置函数、内置类型的方法，或是任何模块和包属性的帮助信息的话，只要你导入了相关属性的必要模块，就可以随时寻求帮助，如help(open)，help(''.strip)，help(time.ctime)等。

1.13 总结

这一章的内容非常丰富，我们希望能迎接所有Django的程序员来到Python的世界。虽然可能没有我们预想的那么全面，但是这毕竟是一本关于Django的书，介绍Python不是我们的主要内容。不过我们还是尽量把Django所需的Python相关知识浓缩在这里。

本章的前三个部分基本上都是和Django有紧密联系的Python语言的材料。最后一部分是一些“软性的技能”：常见的错误，编码风格指导等。希望我们已经展示了足够的内容来帮助你读写一些基本的Python代码，这样你就能顺利地进入Django的世界了。

在下一章里，我们准备手把手地带你在20分钟内完成一个简单的blog应用来感受一下Django。虽然这个blog不像外界商业的系统那样完整，但是它应该能让你感到Django应用开发的迅捷，而且在这一过程中你还能练习一下在本章里新学到的Python技巧。

第2章 Django速成：构建一个Blog

Django自称是“最适合开发有限期的完美Web框架”。所以现在我们来挑战一下，看看到底能在多短的时间里用Django完成一个简单的blog。（关于追求完美的部分我们稍后再细说。）

> **注意**
>
> 本章假设你已经安装了Django。如果还没有的话，请参见附录B。

本章所有工作都是在你选择的shell（bash、tcsh、zsh和Cygwin等）下以命令行方式完成。所以，首先打开终端，键入cd命令进入你PYTHONPATH环境变量所指向的目录。在Linux、Mac OS X、FreeBSD等类UNIX的系统上，你可以用echo $PYTHONPATH命令来查看其内容。在Win32的命令行窗口里，则可以输入echo %PYTHONPATH%。你可以在第1章里读到更多关于路径和安装的内容。

我们建议你在阅读的同时实际动手构建这个blog。如果做不到（例如不在电脑旁边，或者没这个耐性）的话，简单读一下本章也是好的。特别是如果你有其他现代Web框架开发的经验就更好了，因为很多基本的概念都是相似的。

如果你确实打算在电脑上实际操作，同时遇到和你在这里看到的结果不同的情况时，请停下来重新检验刚才的步骤，然后复查之前的两到三步。看看是不是有哪些看上去不重要的地方，或是哪些不能理解的步骤被遗漏掉了。如果还是没发现哪里不对，那就删掉整个项目重新来过。作者在学习Django时就用过这个办法；这样比无助地盯着错误信息看上几个小时要有效率，而且重复那些会导致错误的步骤还能帮你加深印象！

2.1 创建项目

组织Django代码最简单的方式是使用Django的“项目”（project）：一个包含了组成单个网站的所有文件的目录。Django提供了一个叫django-admin.py的命令来帮助创建这样项目的目录。在UNIX上，它的默认安装路径为/usr/bin；在Win32上，它的位置是Python安装目录下的Scripts文件夹，比如C:\Python25\Scripts。无论是哪一个平台，你都要保证django-admin.py在PATH环境变量里以保证它可以从命令行执行。

为blog项目创建一个项目目录的django-admin.py命令是：

```
django-admin.py startproject mysite
```

在Windows上，你需要先打开一个DOS命令窗口。可以通过【开始→程序→附件→命令提示符】来启动这个窗口。另外，你会看到C:\WINDOWS\system32这样的提示符，而不是$符号。

现在我们来看看这个命令为你创建的目录里有点什么内容。在UNIX上它看起来是这样的：

```
$ cd mysite
$ ls -l
total 24
-rw-r--r--    1 pbx  pbx     0 Jun 26 18:51 __init__.py
-rwxr-xr-x    1 pbx  pbx   546 Jun 26 18:51 manage.py
-rw-r--r--    1 pbx  pbx  2925 Jun 26 18:51 settings.py
-rw-r--r--    1 pbx  pbx   227 Jun 26 18:51 urls.py
```

如果你是在Win32平台上进行开发，那么打开资源管理器窗口你可以看到图2.1这样的文件夹，这里我们所创建的目录是C:\py\django，项目的内容将会存放在这里。

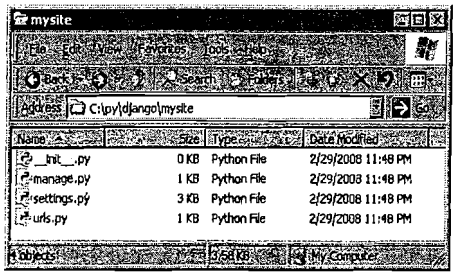


图2.1 Win32上的mysite文件夹

注意

如果你精通Python的话，你一定知道__init__.py会把这个项目目录变成一个Python的包（package）——相关Python模块的一个集合。这让我们可以用Python的“点记号”（dot-notation）来指定项目中的某个部分，比如mysite.urls。（你可以阅读第1章，以获取更多关于包的内容。）

除了__init__.py，startproject命令还创建了以下三个文件：

- manage.py文件是一个同这个Django项目一起工作的工具。你可以从它在目录列表中的权限里看到它是可以执行的。我们马上就会用到它。
- settings.py文件包含了项目的默认设置。包括数据库信息、调试标志以及其他一些重要的变量。你项目里安装的任何应用都可以访问这个文件。稍后在本章进行过程中我们会展示其更多的作用。
- urls.py文件在Django里叫URLconf，它是一个将URL模式映射到你应用程序上的配置文件。URLconf是Django里非常强大的一个特性。

注意

startproject命令创建的所有文件都是Python的源码文件。这里没有XML、.ini文件，或是任何时髦的配置语法。Django追求的是尽可能地保持“纯Python”这一理念。这让你在拥有诸多灵活性的同时无需在框架里引入任何复杂性。例如，如果你想让你的settings文件从其他文件导入设置，或是计算一个值而避免硬编码的话都可以畅行无阻——因为它就是Python而已。

2.2 运行开发服务器

到这里，你还没有构建完blog应用，不过Django为你提供了一些可以就地使用的方便。其中最好用的就是Django的内置Web服务器了。这个服务器不是用来部署公共站点，而是用来做快速开发的。其优点在于：

- 不需要安装Apache、Lighttpd，或是其他任何实际生产所需的Web服务器软件——如果你在一台新的服务器或是没有服务器环境的开发机上，或是就想实验一下的话，那就非常方便了。
- 它会自动检测到你对Python源码的修改并且重新加载那些模块。相比每次修改代码后都要手动地重启Web服务器可是大大地节约了不少时间，这对绝大多数Python的Web服务器设置来说都是很有必要的。
- 它知道如何为admin应用程序寻找并显示静态的媒体文件，所以你就可以直接使用它。

运行开发服务器（或“dev”）简单到只需要一个命令就行了。这里我们要用到项目里的manage.py工具，这是一个简单的包裹脚本，能直接告诉django-admin.py去读入项目特定的settings文件。启动dev的命令如下：

```
./manage.py runserver        # or ".\manage.py runserver" on win32
```

在Win32平台你应该能看到类似如下的输出，只是要退出时要按下的快捷键是Ctrl+Break而不是Ctrl+C。

```
Validating models...
0 errors found.

Django version 1.0, using settings 'mysite.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

在浏览器里输入这个链接，你会看到Django的“It worked!”页面，如图2.2所示。

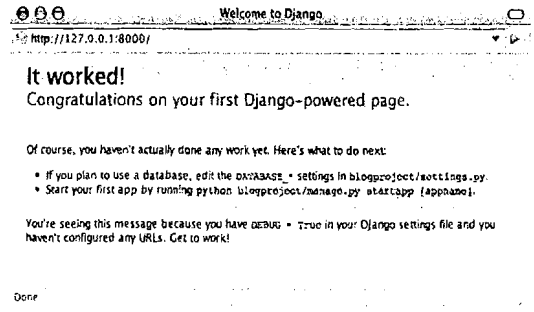


图2.2 Django初始的“It worked!”页面

同时，如果你查看终端的话，你会看到dev服务器记录下了你的GET请求。

```
[07/Dec/2007 10:26:37] "GET / HTTP/1.1" 404 2049
```

log从左到右有4个部分：时间戳、请求、HTTP状态码，以及字节数。（你看到的字节数或

许会有所不同。）这里状态码404（“Not Found”）是因为还没有为项目定义任何URL。而“It worked!”页面只是Django委婉地告诉你这一点的方式。

> **提示**
>
> 如果服务器不工作的话，请重新检查每一个步骤。不要怕麻烦！有时候直接删掉整个项目，然后重新开始可能反而比劳心劳力地检查每一个文件，每一行代码要来的简单。

如果你成功启动服务器后，我们就可以来设置你的第一个Django应用了。

2.3 创建Blog应用

有了项目以后，就可以在它下面创建应用（按Django的说法是“app”）了。我们再次使用manage.py来创建这个blog app。

```
./manage.py startapp blog   # or ".\manage.py startapp blog" on win32
```

这样就完成项目的创建了。现在我们在项目的目录下有了一个blog目录。这是它的内容，首先是UNIX下的，接着是Windows资源管理器里的截图（图2.3）。

```
$ ls -l blog/
total 16
-rw-r--r--   1 pbx  pbx   0 Jun 26 20:33 __init__.py
-rw-r--r--   1 pbx  pbx  57 Jun 26 20:33 models.py
-rw-r--r--   1 pbx  pbx  26 Jun 26 20:33 views.py
```

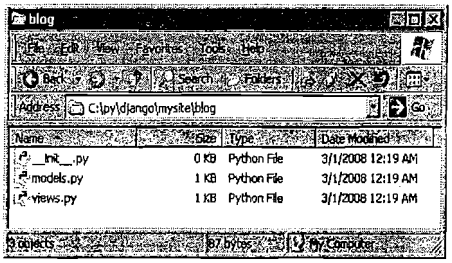


图2.3 Win32上的mysite\blog文件夹

和项目一样，app也是一个包。现在models.py和views.py里还没有真正的代码，它们只是先占住位子而已。实际上对我们这个简单的blog来说，我们根本用不着views.py文件。

要告诉Django这个app是项目里的一部分，你需要去编辑settings.py文件（也称之为“配置文件”）。打开配置文件并在文件尾部找到INSTALLED_APPS元组。把你的app以模块的形式添加到元组里，就像这样（注意结尾的逗号）：

```
'mysite.blog',
```

Django用INSTALLED_APPS来决定系统里不同部分的配置，包括自动化的admin应用以及测试框架。

2.4 设计你的Model

现在我们来到了这个基于Django的blog应用的核心部分：models.py文件。这是我们定义blog数据结构的地方。根据DRY原则，Django会尽量利用你提供给应用程序的model信息。我们先来创建一个基本的model，看看Django用这个信息为我们做了点什么。

用你最习惯的编辑器（如果它能支持Python模式就最好）打开models.py文件。你会看到这样的占位文本：

```
from django.db import models

# Create your models here.
```

删掉注释，加入下列代码：

```
class BlogPost(models.Model):
    title = models.CharField(max_length=150)
    body = models.TextField()
    timestamp = models.DateTimeField()
```

这是一个完整的model，代表了一个有3个变量的“BlogPost”对象。（严格说来应该有4个，Django会默认为每个model自动加上一个自增的、唯一的id变量。）

这个新建的BlogPost类是django.db.models.Model的一个子类。这是Django为数据model准备的标准基类，它是Django强大的对象关系映射（ORM）系统的核心。此外，每一个变量都和普通的类属性一样被定义为一个特定变量类（field class）的实例。这些变量类也是在django.db.models里定义，它们的种类非常多，从BooleanField到XMLField应有尽有，可不止这里看到的区区三个。

2.5 设置数据库

如果你没有安装运行数据库服务器的话，我们推荐使用SQLite，这是最快最简单的办法。它的速度很快，被广泛接受，并且将它的数据库作为一个文件存放在文件系统上。访问控制就是简单的文件权限。更多关于在Django里使用数据库的信息请参阅附录B。

如果你安装了数据库服务器（PostgreSQL、MySQL、Oracle和MSSQL），并且希望用它们而非SQLite的话，你需要用相应的数据库管理工具为Django项目创建一个新的数据库。在这里我们把数据库取名为“djangodb”，不过你可以选用任何喜欢的名字。

当你有了一个（空的）数据库以后，接下来只需要告诉Django如何使用它即可。这就需要用到项目的settings.py文件。

使用数据库服务器

很多人都选用PostgreSQL或MySQL这样的关系数据库和Django配合使用。这里有6个相关的设置（虽然你可能只要两个就够了）：DATABASE_ENGINE、DATABASE_NAME、DATABASE_HOST、DATABASE_PORT、DATABASE_USER和DATABASE_PASSWORD。它

们的作用从名字上就能看的出来。你只需在相应的位置填入正确的值就可以了。例如，为MySQL的设定看上去是这样的：

```
DATABASE_ENGINE = "mysql"
DATABASE_NAME = "djangodb"
DATABASE_HOST = "localhost"
DATABASE_USER = "paul"
DATABASE_PASSWORD = "pony"  # secret!
```

> **注意**
>
> 我们没有指定DATABASE_PORT是因为只有当你的数据库服务器运行在非标准的端口上时才需要那么做。比如，MySQL服务器的默认端口为3306。除非你修改了这个设置，否则根本不用指定DATABASE_PORT。

要知道创建新数据库和（数据库服务器要求的）数据库用户的细节，请参阅附录B。

使用SQLite

SQLite非常适合测试，甚至可以部署在没有大量并发写入的情况下。因为SQLite使用本地文件系统作为存储介质并且用原生的文件系统权限来做访问控制，像主机、端口、用户或密码这种信息一律统统不需要。因此Django只要知道以下的两个设置就能使用你的SQLite数据库了。

```
DATABASE_ENGINE = "sqlite3"
DATABASE_NAME = "/var/db/django.db"
```

> **注意**
>
> 当SQLite配合Apache这样真正的Web服务器一起使用时，你需要确认拥有Web服务器进程的账号也拥有对数据库文件以及包含数据库文件目录的写权限。在我们这里用dev服务器做开发的情况下，这通常不是一个问题，因为运行dev服务器的用户（你）同时也拥有项目的文件和目录。

SQLite在Win32平台上也是非常受欢迎的选择之一，因为它是随同Python一同免费发布的。假设我们已经为项目（和应用程序）创建了C:\py\django目录，现在再来创建一个db目录。

```
DATABASE_ENGINE = 'sqlite3'
DATABASE_NAME = r'C:\py\django\db\django.db'
```

如果你不太熟悉Python的话，你会注意到这个例子和前一个例子的不同之处，之前我们在sqlite3上用的是双引号，而在Win32的版本里，我们用的是单引号。这个和平台是没有关系的——Python没有字符类型，所以单引号和双引号都是被等同对待。只要保证你用同样的引号引用字符串就可以了！

你还应该注意到了文件夹名字前的小写“r”。如果你没有读过第一章，那么你应该知道它的意思是这个对象是一个raw字符串，或者说它把字符串里所有字符都逐字保留下来，不对任何特殊字符组合做转义。例如，\n通常代表一个新行，但是在raw字符串里，它（字面上）代

表了两个字符：一个反斜杠和一个小写n。所以raw字符串的作用（特别是对这里的DOS文件路径来说）就是告诉Python不要转义任何特殊字符（如果有的话）。

创建表

现在你可以告诉Django用你提供的连接信息去连接数据库并且设置应用程序所需的表。命令很简单：

```
./manage.py syncdb          # or ".\manage.py syncdb" on win32
```

在Django设置数据库的过程中你会看到类似这样的输出：

```
Creating table auth_message
Creating table auth_group
Creating table auth_user
Creating table auth_permission
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table blog_blogpost
```

当你执行syncdb命令时，Django会查找INSTALLED_APPS里的每一个models.py文件，并为找到的每一个model都创建一张数据库表。（在稍后我们碰到华丽的多对多关系时会有例外，但是对这个例子来说是正确的。如果你用的是SQLite，你还可以看到django.db数据库会在你指定的位置上被创建出来。）

INSTALLED_APPS里的其他默认条目也都拥有model。manage.py syncdb的输出就确认了这一点，你可以看到Django为每个app都创建了一个或多个表。

但这还不是syncdb命令输出的全部。你还会被问到一些和django.contrib.auth app有关的问题。

```
You just installed Django's auth system, which means you don't have any superusers
defined.
Would you like to create one now? (yes/no): yes
Username (Leave blank to use 'pbx'):
E-mail address: pb@e-scribe.com
Password:
Password (again):
Superuser created successfully.
Installing index for auth.Message model
Installing index for auth.Permission model
```

现在你在auth系统里建立了一个超级用户（就是你自己）。这在等一会我们加入Django的自动admin应用时就会变得很方便。

最后，这一过程还包裹了一些和特性fixture有关的代码，在第4章里我们在讨论这个。这让你在一个新建立的应用里预先载入数据。我们在这里没有用到这个特性，所以Django会跳过它。

```
Loading 'initial_data' fixtures...
No fixtures found.
```

到此数据库的初始化就完成了，下次你再对这个项目运行syncdb命令（只要你添加了应用或是model时就要执行）时，就不会再看到那么多输出了，因为它不需要再次设置表或者提示你创建超级用户了。

2.6 设置自动admin应用

自动化的后台应用程序admin称得上是Django“皇冠上的明珠”。任何对为Web应用创建简单的“CRUD”（Create，Read，Update，Delete）接口感到厌倦的人来说，这绝对是喜从天降。我们会在第11.1节里深入讨论admin。现在我们只要拿来直接用就好了。

由于这不是Django的必要组件，你必须在settings.py文件里指定你要使用它——就和指定blog app一样。打开settings.py并在INSTALLED_APPS元组里的'django.contrib.auth'下面添加这样一行。

```
'django.contrib.auth'.
'django.contrib.admin',
```

每次往项目里添加新的应用后，你都要运行一下syncdb命令确保它所需的表已经在数据库里创建了。在这里可以看到在INSTALLED_APPS里添加admin app并运行syncdb命令会往我们的数据库里加入一个新的表：

```
$ ./manage.py syncdb
Creating table django_admin_log
Installing index for admin.LogEntry model
Loading 'initial_data' fixtures...
No fixtures found.
```

设置完app以后，我们需要为它指定一个URL这样才能访问它。你应该已经注意到了在自动生成的urls.py中的这样两行代码。

```
# Uncomment this for admin:
#    (r'^admin/', include('django.contrib.admin.urls')),
```

删掉第二行开头的#号（也可以同时删掉第一行）并保存文件。这样你就告诉Django去加载默认的admin站点，这是被用于contrib admin应用程序的一个特殊对象。

最后，你的应用程序需要告诉Django要在admin窗口里显示哪一个model以供编辑。要做到这一点很简单，只要定义之前提到的默认admin站点，并向其注册BlogPost model就行了。打开mysite/blog/models.py文件，确认导入了admin应用，然后在最后加上一行注册model的代码。

```
from django.db import models
from django.contrib import admin

class BlogPost(models.Model):
    title = models.CharField(max_length=150)
    body = models.TextField()
    timestamp = models.DateTimeField()

admin.site.register(BlogPost)
```

这里admin的简单使用只不过是冰山一角，它还可以通过为给定model建立一个特殊的Admin类来指定许多不同和admin有关的选项，然后向那个类注册model。这个我们稍后再说，在后面的章节里你还会看到admin的高级使用，特别是本书第三部分和第四部分。

2.7 试用admin

到这里我们已经在Django里设置了admin app并向其注册了model，现在是时候试一试它了。再次运行manage.py runserver命令，随后在浏览器里输入http://127.0.0.1:8000/admin/。（你的dev服务器可能不是这个地址，没关系，只要在后面加上admin/就可以了。）你应该会看到一个登录窗口，如图2.4所示。

输入你之前创建的“超级用户”名字和密码。登录后，你就可以看到admin的主页了，如图2.5所示。

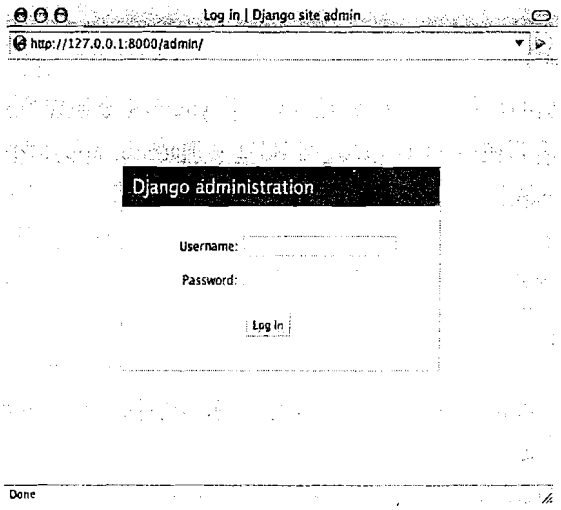


图2.4　admin登录页面

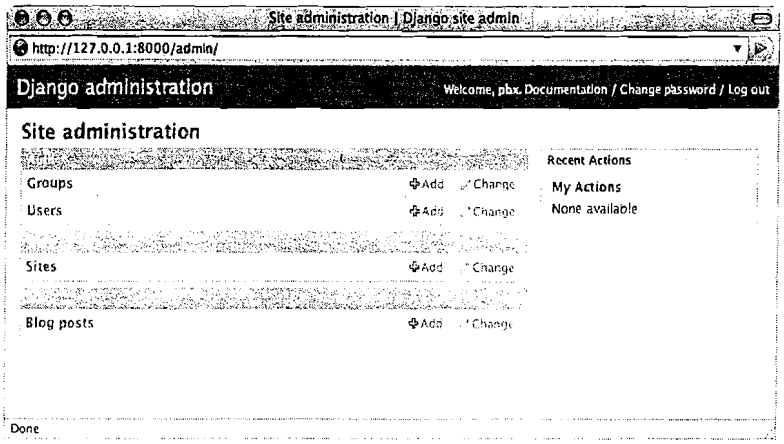


图2.5　admin主页

本书稍后会解释这个界面，现在只需确认你的应用程序Blog和截图里一样出现在屏幕上。如果没有的话，请重新检查之前的步骤。

> **提示**
>
> 三个最常见的“我的app没有显示在admin里”的原因是：1）忘记向admin.site.register注册你的model类；2）models.py里有错误；以及3）忘记在settings.py中的INSTALLED_APPS里添加app。

有没有内容的blog么？点击Blog Posts右侧的Add按钮。Admin会显示一个表单让你添加新的帖子，如图2.6所示。

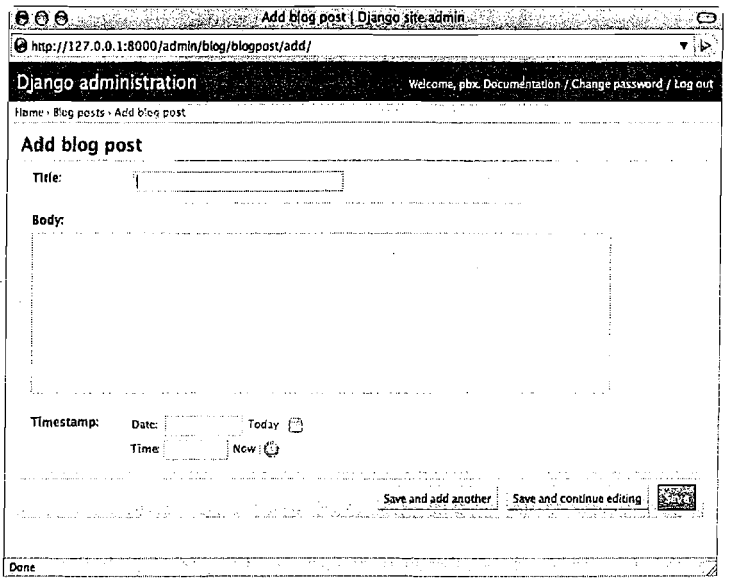


图2.6 通过admin添加新内容

给你的帖子取一个名字然后往里填一点内容。至于时间戳，你可以点击Today和Now的快捷链接来获取当前的日期和时间。你还可以点击日历或时钟标志来方便地选择时间。

完成之后，点击Save按钮保存。你会收到一条确认消息（“The blog post 'BlogPost object' was added successfully.”）和一个列出你所有blog帖子的列表——目前只有一篇而已，如图2.7所示。

为什么帖子有“BlogPost object”这么难看的名字？Django的设计是希望能灵活地处理任意类型的内容，所以它不会去猜测对于给定的内容什么变量才是最合适的。在第三部分的例子里，你会看到如何为你对象的默认标签指定一个特定变量，或是特别定制的字符串。

现在点击右上角的“Add Blog Post +”按钮添加另一篇不同内容的帖子。当你回到列表视图里，你会看见页面上加入了另一个BlogPost。如果你刷新页面或是离开应用程序再回来的话，输出不会有任何变化——你一定不会喜欢看见所有条目都被标为“BlogPost object”，如图2.8

所示。你不是第一个这么想的人，“总有办法让它看起来顺眼一点吧。”

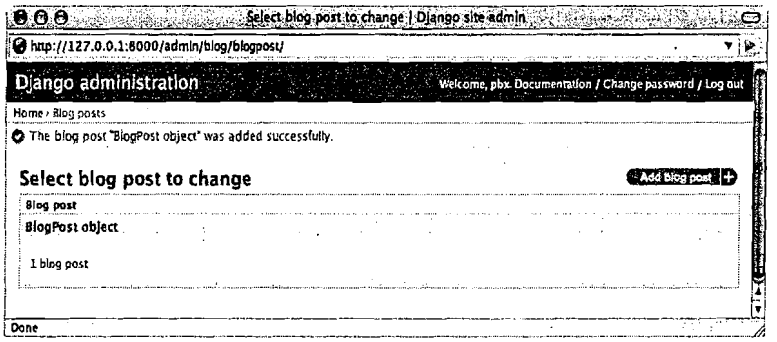


图2.7　成功地保存你的第一篇blog

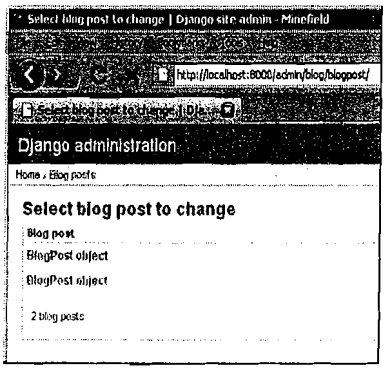


图2.8　不太有用的小节页面

不过我们不需要等到那个时候来清理admin视图里的这些列表显示。之前我们通过很少的配置就激活了admin工具，即向admin app注册我们的model。现在只要额外的两行代码以及对注册调用的一些修改，我们就可以让列表看起来更漂亮更有用。更新你的mysite/blog/models.py文件，添加一个BlogPostAdmin类，并将它加到注册代码那一行里，于是现在models.py看起来是这样的：

```
from django.db import models
from django.contrib import admin

class BlogPost(models.Model):
    title = models.CharField(max_length=150)
    body = models.TextField()
    timestamp = models.DateTimeField()

class BlogPostAdmin(admin.ModelAdmin):
    list_display = ('title', 'timestamp')

admin.site.register(BlogPost, BlogPostAdmin)
```

开发服务器会注意到你的修改并自动重新加载model文件。如果你去看一下命令行shell，

就能看到这些效果。

刷新一下页面，现在你就可以看到一个更有意义的页面了，它是根据你添加到BlogPostAdmin类里list_display变量来生成输出的，如图2.9所示。

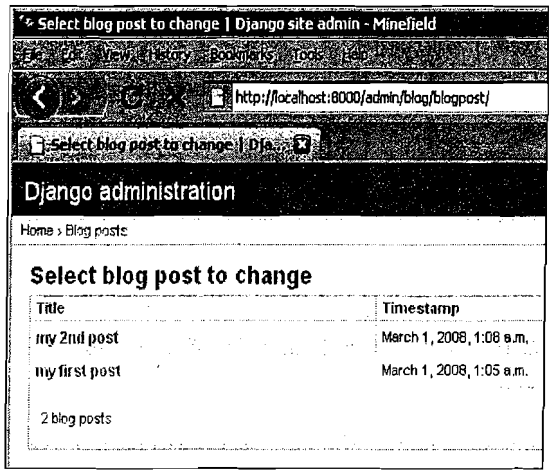


图2.9 这样就好多了

试着点一下Title和Timestamp列的标题——每一个都影响了你的条目是如何排序的。例如，点一下Title会按照升序排列标题，再点一下则变成降序排列。

Admin还有很多有用的特性只需一到两行代码就可以激活：搜索、自定义排序、过滤等。就像我们多次提到的，在本书第三和第四部分里会更详细地展示这些内容。

2.8 建立Blog的公共部分

完成我们应用的数据库部分和admin部分后，现在来看看面向公众的页面部分。从Django的角度来说，一个页面具有三个典型的组件：

- 一个模板（template），模板负责将传递进来的信息显示出来（用一种类似Python字典的对象Context）。
- 一个视图（view）函数，它负责获取要显示的信息，通常都是从数据库里取得。
- 一个URL模式，它用来把收到的请求和你的视图函数匹配，有时也会向视图传递一些参数。

创建模板

Django的模板语言相当简单，我们直接来看代码。这是一个简单的显示单个blog帖子的模板：

```
<h2>{{ post.title }}</h2>
<p>{{ post.timestamp }}</p>
<p>{{ post.body }}</p>
```

它就是一个HTML（虽然Django模板可以用于任何形式的输出）加上一些大括号里的特殊模板标签。这些是变量标签（variable tag），用于显示传递给模板的数据。在变量标签里，你可以用Python风格的dotted-notation（点记号）来访问传递给模板的对象的属性。例如，这里假设你传递了一个叫“post”的BlogPost对象。这三行模板代码分别从BlogPost对象的title、timestamp和body变量里获取了相应的值。

现在我们稍微改进一下这个模板，通过Django的for模板标签让它能显示多篇blog帖子。

```
{% for post in posts %}
<h2>{{ post.title }}</h2>
<p>{{ post.timestamp }}</p>
<p>{{ post.body }}</p>
{% endfor %}
```

原来的3行没有动，我们只是简单地增加了一个叫做for的块标签（block tag），用它将模板渲染到序列中的每个元素上。其语法和Python的循环语法是一致的。注意和变量标签不同，块标签是包含在{% ... %}里的。

把这5行模板代码保存到文件archive.html里，然后把文件放到你的blog app目录里的templates目录下。这个文件的路径即为：

```
mysite/blog/templates/archive.html
```

模板本身的名字是随意取的（叫它foo.html也没问题），但是templates目录的名字则是强制的。Django在默认情况下会在搜索模板时逐个查看你安装的应用程序下的每一个templates目录。

创建一个视图函数

现在我们来编写一个从数据库读取所有blog帖子的视图函数，并用我们的模板将它们显示出来。打开blog/views.py文件并输入：

```
from django.template import loader, Context
from django.http import HttpResponse
from mysite.blog.models import BlogPost

def archive(request):
    posts = BlogPost.objects.all()
    t = loader.get_template("archive.html")
    c = Context({ 'posts': posts })
    return HttpResponse(t.render(c))
```

先略过import那几行（它们载入了我们需要的函数和类），我们来逐行解释一下这个视图函数。

- 第5行：每个Django视图函数都将django.http.HttpRequest对象作为它的第一个参数。它还可以通过URLconf接受其他参数，你将来会大量用到这个特性。
- 第6行：当我们把BlogPost类作为django.db.model.Model的一个子类时，我们就获得了Django对象关系映射的全部力量。这一行只是使用ORM（对象关系映射，详见第3章和第4章）的一件简单例子，获取数据库里所有BlogPost对象。

- 第7行：这里我们只需告诉Django模板的名字就能创建模板对象t。因为我们把它保存在app下的templates目录里，Django无需更多指示就能找到它。
- 第8行：Django模板渲染的数据是由一个字典类的对象context提供的，这里的context c只有一对键和值。
- 第9行：每个Django视图函数都会返回一个django.http.HttpResponse对象。最简单的就是给其构造函数传递一个字符串。这里模板的render方法返回的正是一个字符串。

创建一个URL模式

我们的页面还差一步就可以工作了——和任何网页一样，它还需要一个URL。

当然我们可以直接在mysite/urls.py里创建所需的URL模式，但是那样做只会在项目和app之间制造混乱的耦合。Blog app还可以用在别的地方，所以最好是它能为自己的URL负责。这需要两个简单的步骤。

第一步和激活admin很相似。在mysite/urls.py里有一行被注释的示例几乎就是我们需要的代码。把它改成这样：

```
url(r'^blog/', include('mysite.blog.urls')),
```

这会捕捉任何以blog/开始的请求，并把它们传递给一个你马上要新建的URLconf。

第二步是在blog应用程序包里定义URL。创建一个包含如下内容的新文件，mysite/blog/urls.py：

```
from django.conf.urls.defaults import *
from mysite.blog.views import archive

urlpatterns = patterns('',
    url(r'^$', archive),
)
```

它看起来和基本的URLconf很像。其中的关键是第5行，注意URL请求里和根URLconf匹配的blog/已经被去掉了——这样blog应用程序就变得可以重用了，它不用关心自己是被挂接到blog/下，或是news/下，还是what/i/had/for/lunch/下。第5行里的正则表达式可以匹配任何URL，比如/blog/。

视图函数archive是在模式元组第二部分里提供的。（注意我们传递的不是函数的名字，而是一个first-class的函数对象。当然用字符串也行，你在后面会看到。）

现在来看看效果吧！开发服务器还在运行中么？如果没有，执行manage.py runserver来启动它，然后在浏览器里输入http://127.0.0.1:8000/blog/。你可以看到一个简单朴素的页面，显示了所有你输入的blog帖子，有标题、发布时间和帖子本身。

2.9 最后的润色

你有好几种方式来继续改进这个基本的blog引擎。我们来讨论几个关键的概念，把这个项

目弄得再漂亮一点。

模板的精确定位

毫不夸张地说，我们的模板实在是太平庸了。毕竟这是一本关于Web编程而不是Web设计的书，美术方面的东西你就自己处理吧，但是模板继承是模板系统里另一个能让你减少工作量的特性，特别是当你的页面风格快速成长的时候。

我们这个模板现在是自给自足的。但是如果我们的站点有一个blog、一个相册和一个链接页面，并且我们希望所有这些都能基于同一个基础风格的话该怎么办？经验告诉我们要是用复制粘贴的办法做出三个几乎完全一样的模板肯定是不行的。在Django里正确的做法是创建一个基础模板，然后在这之上扩展出其他特定模板来。在mysite/blog/templates目录里，创建一个叫做base.html的模板，其内容如下：

```
<html>
<style type="text/css">
body { color: #efd; background: #453; padding: 0 5em; margin: 0 }
h1 { padding: 2em 1em; background: #675 }
h2 { color: #bf8; border-top: 1px dotted #fff; margin-top: 2em }
p { margin: 1em 0 }
</style>
<body>
<h1>mysite.example.com</h1>
{% block content %}
{% endblock %}
</body>
</html>
```

虽然不是完全符合XHTML Strict标准，不过差不多就行了。这里要注意的细节是{% block ... %}标签。它定义了一个子模板可以修改的命名块（named block）。修改archive.html模板，让它引用新的基础模板和它的“content”块，就能在blog app里使用它了。

```
{% extends "base.html" %}
{% block content %}
{% for post in posts %}
<h2>{{ post.title }}</h2>
<p>{{ post.timestamp }}</p>
<p>{{ post.body }}</p>
{% endfor %}
{% endblock %}
```

这里的{% extends ... %}标签告诉Django去查找一个叫base.html的标签，并将这个模板里命名块的所有内容填入到那个模板里相应的块里去。现在你应该可以看到类似图2.10的页面了（当然了，你的blog内容比我这个应该更加精彩）。



图2.10 稍微带点风格的blog

按日期排序

你应该已经注意到blog的帖子不是按照传统的时间倒序排列的。告诉Django这么做非常简单。实际上，有好几种做法可供选择。我们可以在model里加入一个默认的顺序，或者在视图代码里的BlogPost.objects().all()上添加排序功能。在这里修改model会比较好，因为基本上帖子都是按时间倒序排列的。如果在model里设置我们想要的排序方式，Django里任何访问数据的部分都会采用这个排序结果。

设置model默认排序的方法是给它定一个Meta嵌套类，然后设置ordering属性。

```
class Meta:
    ordering = ('-timestamp',)
```

现在看一下blog的首页（/blog/）。最新的帖子应该出现在页面最上方了。字符串“-timestamp”能简洁地通知Django，“对‘timestamp’变量按照降序排列”。（如果省略“-”的话则是按升序排列。）

> **注意**
>
> 千万不要忘了小括号里结尾的那个逗号！它代表这是一个单元素的元组，而不是一个带小括号的字符串。Django在这里要的是一个元组，你可以排序任意数目的变量。如果你在逗号后面加上'title'，并且你有两个相同发布时间的帖子“A”和“B”的话，“A”就会出现在前面。

通过模板过滤器格式化时间戳

虽然时间戳很好用，但是它的ISO8601格式却有点怪异。我们现在用Django模板系统里另一个很酷的特性：过滤器（filter）来把它弄得人性化一点。

由于这是一个表示层的（presentation）细节而非数据结构或是商业逻辑的细节，最适合它的位置应该是模板。打开archive.html文件并修改“post.timestamp”一行。

```
<p>{{ post.timestamp|date }}</p>
```

只需像这样用一个竖杠，或“管道”符号接在变量名后面（大括号内部）就能把过滤器应用于变量之上了。刷新blog首页。现在你可以看到日期的显示更加友好了（“July 7”）。

如果date过滤器的默认风格你也不喜欢的话，你可以传递一个strftime风格的格式化串作为它的参数。不过这里它用的不是Python里time模块的转换代码，而是PHP的date函数的格式化说明。例如，如果你要显示星期几，但是不要显示年份的话，代码就变成下面这个样子了：

```
<p>{{ post.timestamp|date:"l, F jS" }}</p>
```

这个格式化字符串会返回“Friday, July 6th”风格的日期。注意这里不要在冒号两边留有空格——Django的模板引擎对空格敏感。

2.10 总结

我们可以给这个blog引擎不停地加入新特性（很多人就是这么干的！），但是作为体验Django能力的话，希望你已经看的差不多了。在构建这个基本的blog app过程中你已经看到了好几个Django优雅、简洁的特性：

- 内置的Web服务器能让开发工作自给自足，同时它能在代码被修改时自动重新加载你的代码。
- 数据模型的创建采用纯Python方式完成，你不用编写维护任何SQL代码或XML描述文件。
- 自动化的admin应用程序，提供了完整的内容编辑特性，甚至非技术背景的用户也能使用。
- 模板系统，可以用来生成HTML、CSS、JavaScript以及任何文本输出格式。
- 模板过滤器，可以在不解除应用程序商业逻辑的前提下修改表示层的数据显示（比如日期）。
- URLconf系统，在给予你URL设计极大灵活性的同时还能将应用程序特定的URL部分保留在其所属的应用程序内部。

下面是一些可以通过Django内建的特性就能添加到blog上的功能，这能让你有个基本的概念：

- 发布关于最新帖子的Atom或RSS feed（参见第11章）。
- 添加一个搜索功能，让用户能迅速定位包含特定词句的blog帖子（参见第8章里的CMS例子）。
- 用Django的“通用视图”来避免在views.py里编写任何代码（参见第10章里的Pastebin示例）。

到此你已经完成了Django基础的旋风之旅。第3章会总览Django中的关键组件和它们背后的设计哲学，以及简要地复述一下Web开发原则，它们不仅是对Django自身，而且也是对书后几章的经验教训总结。第4章将带你进入框架的细节，你会找到很多之前例子中碰到的关于“怎么样，为什么和什么是等等？”问题的解答。第4章之后你就会对框架有足够的理解，可以继续构建示例应用了：一个内容管理系统（CMS）、一个Pastebin、一个相册和一个基于Ajax技术的“live blog”。

第3章 起　始

与任何大型软件项目一样，Django包含了大量的概念、特性和工具，而且它所要解决的问题（即Web开发）的范围也是相当得广。在开始学习使用Django的细节之前，你先得理解这些问题是什么，以及像Django这样的框架是怎么解决它们的。

这一章将会逐一介绍这些基本概念，首先是对Web的一个总体认识，然后是Web框架模型及其组成部分的解释，最后是Django的创造者对软件开发的一些普遍的看法和理念。上一章里的一些概述内容也能帮助你更好地理解本章的内容。

这里有一个重要的提示：如果你对Web开发相当熟悉的话，这里的一些概念已经有些老套了——但即使是再有经验的Web工程师也可以温故知新嘛。有时候之所以不识庐山真面目，主要还是因为我们的思想都被某种特定的语言或工具束缚住了，要是看问题的时候站的高一点，说不定就柳暗花明了。

牢牢掌握这些基础概念能让你在设计和实现是做出更好的选择。所以，请不要跳过这一章！

3.1 动态网站基础

Web开发理论上来说还是很简单的。用户向Web服务器请求一个文档；Web服务器随即获取或生成这个文档；服务器再把结果返回给用户的浏览器；最后浏览器将这个文档渲染出来。细节虽然各不相同，但是基本上的流程都是如此。我们现在来看看Django里的每一步是怎么样的。

通信：HTTP、URL、请求、响应

HTTP（超文本传输协议）封装了Web页面服务的整个过程，它是Web的基石。由于这个协议是面对客户/服务器端通信之用，所以它主要就是由请求（request，客户端到服务器端）和响应（response，服务器端到客户端）两个部分组成。而两者之间的服务器上发生的事情都不属于HTTP关心的范畴，完全是由服务器软件决定的（看下面）。

请求封装了过程的第一部分——客户端向服务器端要求一个给定的文档。请求的核心就是URL（指向所需文档的“路径”），当然它可以通过一系列方法进一步参数化，让单个的地址或URL展现多种行为。

响应主要是由一个正文（body，通常是Web页面的文本）和相应的包头（header）组成。包头里是关于所需数据的额外信息，例如最后更新的时间，在本地可以缓存多久，内容的类型等等。另外，响应里非HTML的内容可以是纯文本，文档（PDF，Word，Excel等），声音片段等。

Django将请求和响应表示成相对简单的Python对象，用属性来表示其数据，以及用方法来进行更复杂的操作。

数据存储：SQL和关系数据库

粗看起来，Web的作用就是传输数据或内容共享（这里的内容可以是任何东西——blog帖子，金融数据，电子书等）。在早期的Web里，内容都是手工编写的HTML文件，一般都是保存在数据库的文件系统上。这叫做静态（static）内容，因为同样的URL请求返回的信息总是一样的。之前描述的“路径”在这里相对简单，一般都没有参数，毕竟它只不过是指向服务器文件系统上静态内容所在的一个路径而已。不过今时不同往日，现在大多数的内容都是动态的了，因为一个给定URL可以根据参数的不同返回完全不一样的数据。

大部分这种动态的特性是通过将数据保存在数据库里实现的，在数据库里，数据不再是简简单单的一个文本字符串，你可以创建有多个组成部分的数据，并且将它们连在一起来表达其中的相互关系。SQL（结构化查询语言）是用来定义和查询数据库的语言，通常被进一步抽象为一个ORM（对象关系映射），它可以把数据库里的数据映射为面对对象语言里的代码对象。

SQL数据库按照表（table）的形式来组织，每张表则由行（row，例如，条目、对象）和列（column，属性、变量）组成，基本上和数据表格很相似。Django提供了一个强大的ORM机制，Python的类就代表了表，对象代表了其中的每一行，而对象的属性则代表了列。

表示：将模板渲染成HTML和其他格式

Web开发里的最后一个部分就是如何表示或者格式化用户所请求的信息，经由HTTP返回的信息，以及SQL数据库查询所返回的信息。通常，结果都是表示成HTML（超文本标记语言）或更XML一点的XHTML，并且配合负责完成浏览器端功能的JavaScript和视觉效果的CSS（层叠样式表）。再新一点的应用还会用JSON（一种“轻型”数据格式）或XML来完成动态内容。

大多数Web框架都提供了模板语言（template language）来处理要显示的数据，它混合了原始的HTML标签以及一些类似编程的语法来循环对象集合，执行逻辑操作和其他一些动态行为所需的结构。一个简单的例子就是一份静态HTML文档加上一点点逻辑来显示当前登录用户的用户名，或者在没有用户登录的情况下显示一个“登录”的链接。

有些模板系统通过把它自己的命令实现为HTML属性或是标签来试图完整地兼容XHTML，所以文档生成的结果可以按照普通HTML那样来解析。其他一些则更贴近常规的编程语言，有时候还会将程序结构用特殊的符号包裹起来以便阅读和解析。Django的模板语言就属于后面这种。

组合在一起

尽管Web被分成前面说到的三个组成部分，但是有一个关键的地方被忽略掉了：它们之间是如何相互交流的。Web应用程序是怎么根据请求知道要去执行一个SQL查询的，以及它是如何知道要用哪一个模板来渲染结果的呢？

答案是这要看用的是什么工具了：每个Web框架或语言都有不同的方法。不过通常相似的地方总要比不同的地方多，所以虽然下面的两节讲的是Django自己的方式，但是这些概念在其他框架里也可以找得到。

3.2 理解模型、视图和模板

就像你刚看到的那样，Web开发经常被分割成几个核心的组件。在这一节里，我们要进一步拓展这些概念，讨论一些编程方法论，然后总览一下Django是如何实现它们的（并在后面的章节里展示细节是实例）。

分层（MVC）

将（Web或其他的）动态应用程序分层这种思想已经存在很久了，通常都是将其应用在图形化的客户端应用上，学名叫MVC（Model-View-Controller，模型-视图-控制器）范式。即，应用程序被分割成模型（控制数据），视图（定义显示的方法），以及控制器（在两者之间斡旋，并且让用户可以请求和操作数据）。

把一个应用程序以这种风格分成几个部分可以给予程序员足够的灵活性，并且鼓励重用代码。例如，对于一个给定的视图，假设这个模块知道如何图形化数字类型的数据，那么只要中间的胶水代码能把它和数据联系起来，它就可以用在各种不同的数据集上。或者一个特定的数据集可以用多种不同的输出格式来显示，例如刚刚提到的图形化视图，纯文本文件或是可排序的表格等。多个控制器可以根据用户的不同对同一个数据模型做出不同程度的访问控制，或是允许通过GUI应用以及email或是命令行来提交数据。

成功实施MVC架构的关键在于要正确地分割应用程序的不同层次。虽然在某些情况下，在数据模型里存放如何显示它的信息是贪图一些方便，但是却会给将来替换视图带来极大的困难。同样，在图形布局的代码里放置数据库相关的代码会在替换数据库平台的时候让你头痛不已。

Django的办法

Django也遵循了这一分层的原则，不过在做法上却略有不同。首先模型部分保持不变：Django的模型层只负责把数据传入传出数据库。然而Django里的视图却并不是显示数据的最后一步——Django的视图其实更接近MVC里传统意义上的控制器。它们是用来将模型层和表示层（由HTML和Django的模板语言组成，稍后在第6章里会介绍到它）连接在一起的Python函数。按Django开发团队的话来说就是：

我们理解的MVC里，视图的作用是描述将要显示给用户的数据。这不仅仅是数据看上去的外观（look），还包括如何表示数据（present）。视图描述的是你能看哪些数据，而不是怎么看到它。这里面的区别很微妙。

换一种说法，Django把表示层一分为二，视图方法定义了要显示模型里的什么数据，而模板则定义了最终信息的显示方式。而框架自己则担当了控制器的角色——它提供了决定什么视图和什么模板一起响应给定请求的机制。

模型

任何应用程序的基本，不管是不是Web应用，都是它所展现、收集和修改的信息。因此，若将应用程序分层，模型（model）将是最底部的一层，它是基础。视图和模板可以根据数据

进出模型的方式以及表现的形式任意替换，但模型却相对稳定得多。

从设计整个Web应用的角度来说，模型可能是最容易领会却也是最难掌握的部分。在面向对象系统里对一个现实问题进行建模相对来说通常不难，但是对大流量的网站来说，最符合实际的模型却不一定总是最高效的。

模型里有很多潜在的陷阱，其中一个就是在应用程序部署后修改模型的代码。虽然表面上你“只是在修改代码”，但实际上确实在修改底层的数据库模式，这经常会对已经存储在数据库里的数据产生不良的副作用。在后面设计示例应用的章节里，我们遇到很多这类现实问题。

视图

视图（view）组成了Django应用程序里很多（有时候几乎是全部）的逻辑。它们的定义实际上却很简单：它们是链接到一个或多个定义URL上的Python函数，这些函数都返回一个HTTP响应对象。在Django的HTTP机制两端之间要执行什么操作完全都在你的控制之下。实际上，在这一步通常只有一些简单的任务需要完成，例如显示一个或是一列从模型里取得的对象，或者是往模型里添加这样的新对象，加上一些额外的工作诸如检查应用程序用户验证的状态并决定是允许还是拒绝访问。

Django为这类任务提供了很多方便快捷的帮助函数，但是你也可以选择自己编写一切来完全地控制整个过程，或是大量使用这些快捷方式来进行快速建模和开发，抑或是两者结合。灵活和强大两者兼备就是视图的特点了。

模板

我们刚刚说视图的作用是负责显示来自模型的对象。这并非百分之一百的正确。如果说视图方法只是返回一个HTTP响应的话，那么是正确的——你可以在Python里写出一个字符串然后返回它，这样的做法也不算是很落伍。只不过在绝大多数情况下，这样做的效率实在太低，所以在这里引入一个抽象层非常重要。

所以大多数Django程序员都使用它的模板语言来渲染HTML页面。基本上，模板就是一些输出动态值的经过特殊格式化的HTML文本，支持简单的逻辑结构如循环等。当一个视图要返回一个HTML文档时，它通常会指定一个模板，提供给它所要显示的信息，并在响应里使用模板渲染的结果。

虽然HTML是最常见的格式，模板实际上并不一定要它——模板可以生成任何文本格式，如CSV，甚至email消息正文。重要的是它们让一个Django项目能够把数据的表示和决定显示什么数据的视图代码区分开来。

3.3 Django架构总览

本章到目前为止，我们已经讲解了组成实际Django系统的主要架构组件以及外围的一些小配角。现在我们来把它们放到一起，给出一个总览。如图3.1，你可以看到最接近用户的是HTTP通信协议。通过URL，他们可以想Django的Web应用发送请求，并且在Web客户端接受响

应，同时客户端可能还用到了Ajax技术，利用JavaScript来处理一些非顺序性的服务器访问。

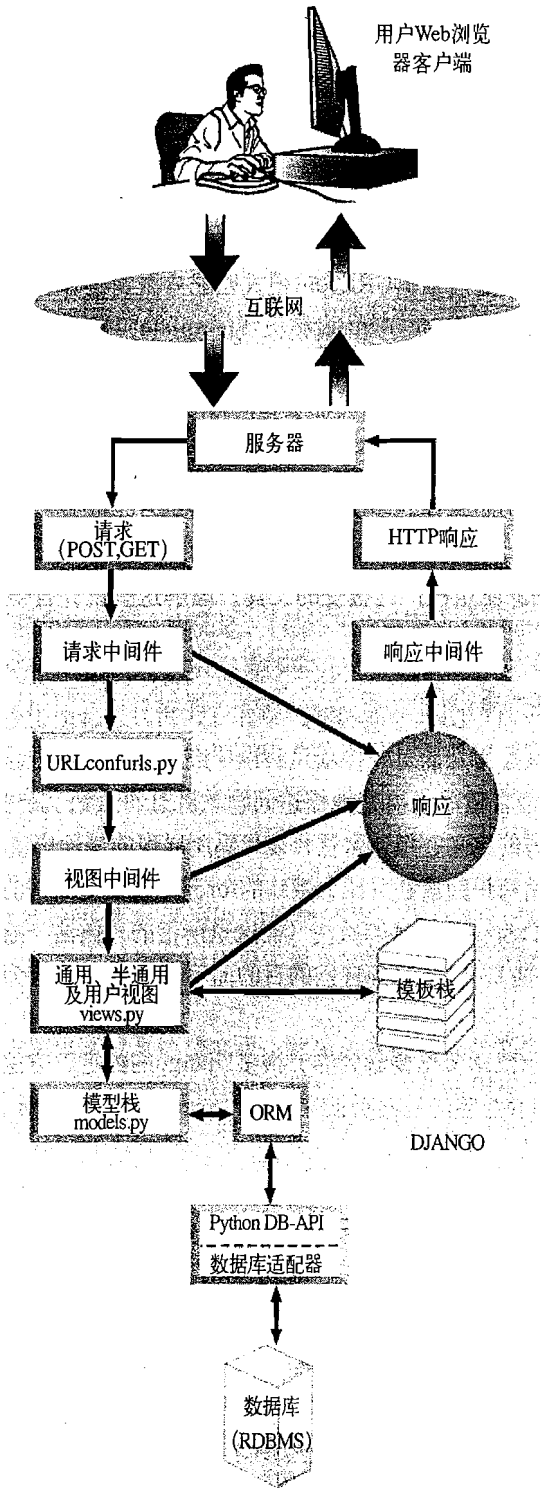


图3.1 Django组件架构图

在另一端（图的底部），你可以看到在你的模型（model）和Django ORM管理下的数据库持久层存储，它们通过Python的DB-API以及作为适配器的数据库客户端库（通常是以C/C++编写并提供一份Python接口）与数据库通信。

最后，两者之间的正是应用程序的心脏，Django。MVC模型在Django的术语里应该是“MTV”。视图（view）是作为控制器通过ORM负责从数据库里创建、更新和删除数据模型，同时根据给定的模板管理最终用户看到的结果。

把所有部分拼起来就是这样的，收到的HTTP请求被Web服务器转发给Django，Django在请求的中间件层接受它们。随后根据URLconf模式匹配分派到适合的视图上去，视图会执行所需工作的核心部分，用模型（model）和/或模板（template）按需要生成响应。随后响应再次穿过中间件层进行最后的处理，最后将HTTP响应返回给Web服务器并转发给用户。

3.4 Django的核心理念

作为一个最初由一支小而精悍的团队所开发的完整的Web框架，Django一直，而且将继续遵循一些特定的设计理念。这完美地体现了其核心团队的经验（以及从某种程度上说的，气质），同时他们也非常乐意使用被广泛接受的工具。理解这些设计哲学能帮助你更好地理解和使用框架。

Django希望尽量Pythonic一点

使用编程语言的社区通常是能对软件项目的设计产生重大影响的因素之一，Django也不例外。Python的用户一般都希望尽量清楚地描述问题，并且符合语言自身的理念，即Pythonic。虽然对这个属于没有官方的定义，但通常这代表代码所展现的属性应该符合语言的各个方面。

这些属性包括有使用精悍的语法（比如默认的for循环或是更简洁的列表推导式）等。即通常对每种简单的任务，只有一种正确的做法（而这里所指的“做法”通常都已经被结合到语言里了，比如字典的get方法），以及明确自己的目的而不要依赖默认行为（例如所有对象方法都要求的self参数）。

如同在第三部分的示范章节里要看到的，很多Django的约定，方法和设计都是或努力接近Pythonic的。因此有Python经验的程序员很容易入门，同时也能帮助新手养成良好的编程习惯。

不要重复自己（DRY）

Pythonic的这一属性值得单列到一节里讲，因为它是一个对几乎所有编程工作来说都非常重要的原则：DRY，不要重复自己。DRY可能是所有编程惯例（programming idiom）最简单的，因为它实在只是一个常识罢了：如果你要修改在多个地方重复出现的信息，你就平白无故给自己制造了两倍（甚至更多）的工作量。

作为DRY的一个例子，要对一些数据进行一些简单的运算，比如对某个人名下的所有银行账户加和。在设计糟糕的系统里，这种加和运算可能出现会在整个代码里的多个位置，显示用户的页面、每个用户详细信息的页面、或者是显示多个用户综合的页面。在Django ORM这样

的系统里，你只需为Person类创建一个sum_accounts方法就可以轻易遵循DRY原则了，这个方法只需要定义一次，然后就能在上面说到的所有地方使用。

虽然在上面这样的简单情况里应用DRY的难度并不大，它却也是最难在任何时候都严格遵守的戒律之一。很多时候它会和其他（Pythonic等）惯例发生冲突，这时就必须做出妥协。不过，大多数时候这都是值得努力接近的目标，而且随着经验的不断丰富遵守它也会变得容易起来。

松耦合与灵活性

Django是一个功能完整的Web框架，提供了所有动态Web应用所需的一切必要组件：数据库访问、请求框架，应用程序逻辑，模板系统等。同时，它也努力保持开放：你可以按需要选择或多或少的Django组件并随时替换为你认为适合的其他工具。

例如，有些用户不喜欢Django的模板系统而比较偏爱Kid或Cheetah这样的选择。Django的视图方法并不一定要使用Django的模板系统，所以你的视图完全可以去加载Kid或是Cheetah，渲染用这些系统编写的模板，并将其作为Django响应对象的一部分返回。

数据库层也是一样。如果用户比较喜欢SQLAlchemy或是SQLObject的话，可以完全无视Django的ORM而转去使用这些工具。相反地，（虽然不太常见）Django的ORM也可以被单独拿出来用于其他项目（甚至非Web应用），只需很少的几个步骤就能让它工作起来。

不过毕竟这样的模块化还是有代价的：有些Django最出挑的特性却一定要把整个项目捏在一起才能使用，比如通用视图方法，它能让你轻易地显示、更新和创建数据库记录。因此，对Python Web开发的新手来说，最好还是先暂时不要关心Django的这种模块化方式。

快速开发

Django在编写的时候就考虑到了快速和敏捷开发。工作在一家快节奏的本地报馆里，Django的核心团队需要一组能在极短的时间里实现功能的工具。开源整个框架并没有改变它擅长这一领域的事实。

Django在不同层面上提供了很多快捷方式。这里最明显的就是刚刚提到的通用视图，它由差不多一打常见的任务组成。兼具灵活和强大的参数化，这些通用视图可以，而且经常，组织其整个网站，完成数据库记录的创建和修改、显示多个对象（面向日期等）以及单个对象的页面等许多功能。只需三个Python文件（站点相关的设置，一个模型声明和一份链接URL到通用视图的映射）以及一些HTML模板，就可以在几分钟到几个小时里完成一个网站了。

Django在Python视图层上也为很多常见任务提供了快捷的方法，所以当通用视图无法满足程序员的需要时，他们依然能避免自己动手。这样的快捷方式用数据字典渲染模板，获取数据库对象，当数据不存在时返回HTTP错误，以及处理表单等。

结合了Python的灵活、简洁和强大，这些快捷方式让程序员在得以尽快地完成项目，解决特定问题的同时，无需担心乏味的重复劳动，或所谓的“胶水代码”。

3.5 总结

这一章涵盖了很多基础内容：什么是Web开发，Django及类似框架创建网站的方式，以及推动Django自身开发和设计背后的理念。不管你还记得多少，我们希望你从这一章里获得了一些什么。

到此，你应该已经了解了开发Web应用的基础背景，以及一个典型的Web框架背后的理论及其组织方式。在第二部分里，我们会详细讲解如何使用Django，讨论它用到的各种类、函数和数据结构，并且向你展示更多代码示例来帮助你理解它们。

第二部分 深入Django

第4章 定义和使用模型
第5章 URL、HTTP机制和视图
第6章 模板和表单处理

第4章 定义和使用模型

在第3章中解释过，数据模型通常是Web应用程序的基础，从这里开始探索Django开发的细节非常合适。虽然这一章有两个主要的部分（定义模型和使用模型），但它们不是独立的小节，而是更多地交织在一起。我们需要在定义模型时就考虑好怎么使用它们，以便生成最有效的类和关系布局。当然，如果没有理解怎么定义和为什么定义模型的话，你是无法充分利用它们的。

4.1 定义模型

Django的数据库模型层大量使用了ORM（对象关系映射），理解这个设计背后的原因，以及这个方法优缺点是非常重要的。所以，在一节里我们首先解释一下Django的ORM，然后我们在深入讲解模型变量（model field）的细节，模型类（model class）之间可能的关系，以及通过模型类元数据（model class metadata）来定义模型的特定行为或是激活和自定义Django的admin应用。

为什么使用ORM

Django和其他大多数现代Web框架（以及很多其他应用程序开发工具）一样，依赖于一个强大的数据访问层，它试图将底层的关系数据库和Python的面向对象特质联系起来。这些ORM在开发社区里依然是一个争议很大的话题，有支持的也有反对的。Django在设计之初就决定要使用ORM，我们有4个使用它的理由，特别是Django自己的实现。

封装有用的方法

本章稍后会介绍到的Django模型对象是定义一组变量的首选方式，而变量通常对应的是数据库的列。这就为连接关系数据库和面向对象概念跨出了第一步也是坚实的一步。你可以请求一个id为5的Author对象并检查它的author.name值，而不必去写SELECT name FROM authors WHERE id=5这样的SQL查询——比较起来这种类型的数据接口要Pythonic得多了。

而且，模型对象能给那个简陋的例子增加许多额外的价值。和很多其他系统一样，Django

的ORM可以让你定义任何实例方法，这样很多事情就变得有用起来。例如：

- 你可以定义只读的变量或属性组合，有时候也称之为“数据集合”（data aggregation）或者“计算属性”（calculated attribute）。例如，一个具有count和cost属性的Order对象可以暴露一个计算两者乘积的total属性。常用的面对对象设计模式因此变得相当容易——比如外观模式（facades），或是代理模式（delegation）等。
- Django的ORM允许重写内置的数据库修改方法，例如保存和删除对象。这样你就可以在数据被保存到数据库之前轻易地对它进行任何操作，或者在删除一条记录之前确保特定的清理工作都会先被调用，无论删除在什么地方以及如何发生的。
- 和编程语言集成（拿Django来说，自然就是Python）通常比较简单，可以让你的数据库对象和特定的接口或API更加一致。

可移植性

由于自身的特性（作为应用程序和数据库之间的代码层），ORM通常具有很好的可移植性。大多数ORM平台都支持数种数据库后台，Django也一样。在本书编写的时候，Django模型层的代码可以运行在PostgreSQL、MySQL、SQLite，以及Oracle——随着更多数据库后台插件的开发，这张表会越来越长。

安全性

因为使用ORM后很少再有机会需要自己执行SQL查询，所以你不必再担心由不合格的或是保护性很差的查询字符串所导致的问题，例如SQL注入攻击等。ORM还提供了一个智能化地引用和转义输入变量的核心机制，让你不用再花很多时间处理这种细枝末节。这类优点在模块化和层次化的软件（比如MVC框架就是一个好例子）里十分常见。当所有负责一个特定问题领域的代码完备地组织在一起的时候，它通常能为你节省很多时间同时还增强了整体的安全性。

表现力

虽然和模型的定义并没有直接的联系，然而使用ORM最大的一个好处（和直接编写SQL相比，当然也是最大的不同之处）就是从数据库获取记录时使用的查询语法。高级一点的语法不仅更容易编写，而且这种带入到Python领域里的查询机制更带来了一系列有用的技巧和方法。例如，直接循环一个数据结构原本在SQL查询是相当笨拙的，相反现在却简洁得多。同时还可以回避一些本来是无可避免的讨厌的字符串操作。

Django丰富的变量类型

Django的模型拥有多种不同的变量类型，有些和它们在数据库里的实现比较接近，有些则是为Web表单界面而考虑设计。基本上所有类型都属于两者之间。完整的列表可以在Django的官方文档里找到，不过这里我们比较性地列出一些最常用的用户变量。首先，我们给出一个基本的Django模型定义。

```
from django.db import models


class Book(models.Model):
    title = models.CharField(max_length=100)
```

```
author = models.ForeignKey(Author)
length = models.IntegerField()
```

这个例子的作用相当明显：我们为book创建了一个简单的模型，它包含了好几个和数据库相关的概念。不是很复杂（通常这类给图书分类的工作要考虑的东西可不局限于书名，作者和页数）但是放在这里够用了。而且它完全可以直接使用，把它扔到Django的models.py文件里，只需很小的修改就能完成一个图书分类的应用了。

你可以看到，Django用Python的类来表示对象，而对象则通常映射到SQL中的表，对象的属性则是表中的列。这些属性自身也是对象，它们都是Field类的子类。前面说到，它们中的有些类型和SQL的列类型很相似，其他的则都提供了某种程度的抽象。我们来看一些特定的Field子类。

- CharField和TextField：这可能是你会遇到的最常用的变量类型了，这两个类型基本上是一样的——作用都是保存文本。区别在于CharField是定长的，而TextField的长度则可以是无限的。具体使用哪一个要看需要，包括数据库的全文搜索能力或是高效存储的需求。
- EmailField、URLField和IPAddressField：这三个变量其实就是CharField加上一点额外的验证。它和CharField一样存储在数据库里，但是包含了验证代码来保证它们的值分别是有效的E-mail地址、URL和IP地址。你也可以很轻松地在模型变量上加入验证来创造出你自己的“变量类型”，这些类型和Django的内置类型是平等的。（参见第6章及第7章里关于验证的更多细节。）
- BooleanField和NullBooleanField：BooleanField大多适用于你要存储True或False值的场景，但是有时候你还不知道要存储哪个值的时候就需要更大的回旋余地——这时变量可以是空的或null的，所以就出现了NullBooleanField。这个特性体现了有时候在为数据建模的时候不但要在语义层面考虑问题，同时还需要有技术层面的考量——即不仅要关心数据是如何存储的，还要关心它的意义。
- FileField：FileField是最复杂的变量之一，因为大多数时候所有和它有关的工作和数据库一点关系都没有，相反主要是由于框架本身的需求。FileField只在数据库里保存了一个文件的路径，和它的近亲FilePathField很像，但是它允许用户从浏览器上传一个文件，并将它保存在服务器上。它还在模型对象里提供了访问网页URL的方法来接受要上传的文件。

这些只是Django的模型定义里常见的变量类型，当新版的Django发行时，新的变量类型也会时不时地被添加进来或是更新。要查看完整最新的模型变量类的列表以及可以进行的操作，可以查阅Django的官方文档。你还会在本书中看到很多这些变量，比如小代码段或是第三部分中的示例应用。

主键和唯一性

关系数据库定义里一个常见的概念就是主键（primary key），这个变量保证在整张表中（按Django ORM的说法就是在整个模型中）的唯一性。这些主键通常都是自增的整数因为这种确保表里面每条记录都有一个唯一值的方法既简单又有效。

它们还可以方便地在有关联的表之间被别人引用（稍后几节里会讲到）——如果给定一个

ID为5的Book对象并且保证它是唯一拥有这个ID的Book时，引用“第5号book”就不会有歧义了。

因为这种主键的使用相当普遍，除非你明确指定，否则Django会为你自动生成。所有没有显式定义主键变量的模型都会被指定一个id属性，其类型为Django的AutoField（一个自增的整数）。AutoField和普通整数一样，而它们在数据库中的类型则是根据你使用的数据库后台不同而不同。

如果你希望有更多的控制主键，只需在模型的某个变量上指定primary_key=True就行了，这个变量会取代id（这时id会被忽略掉）成为这张表的主键。这意味着变量的值必须是唯一的，所以指定一个姓名这样的字符串或是其他一些标识符可不是什么好主意，除非你有110%的把握永远不会发生重复的情况。

说到重复，还有一个类似的参数，它可以应用到模式里任何变量上：unique=True。这也能保证那个变量的唯一性，不过你不需要把它当作主键。

模型之间的关系

定义模型对象之间关系的能力通常是关系数据库最大的卖点之一（证据就是它的名字——关系），同时这也是ORM之间相互竞争的区域。Django目前的实现主要还是围绕着数据库展开，确保关系是在数据库层而不是在应用层上定义。然而，由于SQL值提供了一种显式组织关系的方法（外键），我们有必要再加入一些抽象来表示更复杂的关系。我们先来看看外键的定义，然后看它是如何像搭积木一样构建起其他关系类型的。

外键

主键的概念相对比较简单，所以Django对它的实现也比较直观。它们表示成一个Field的子类ForeignKey，其主要参数就是它要引用的模型类，就像下面的例子：

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author)
```

请注意你需要把被引用的类定义在前面，否则Author变量名是无法在Book类的ForeignKey变量里使用的。不过，你可以用字符串来代替，如果类是在同一个文件里定义的，那就只需要类名，否则的话就需要用点记号（比如，'myapp.Author'）。这里在ForeignKey里使用字符串重新组织编写上面的例子：

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey("Author")

class Author(models.Model):
    name = models.CharField(max_length=100)
```

如果要引用自己的话，也可以把ForeignKey指定为字符串'self'。这在定义层次结构（例如，

一个Container类可以定义一个parent属性来嵌套另一个Container）等类似场景（比如Employee类可以具有类似supervisor或是hired_by这样的属性）的时候很常用。

虽然ForeignKey只定义了关系的一端，不过另一端却能根据关系追溯回来。外键从技术上说是一个“多对一”的关系，即可以有多个子对象引用同一个父对象。因此子对象只有一个父对象的引用，而父对象却能访问到一组子对象。拿上面的例子来说，你可以这样使用Book和Author的实例：

```
# Pull a book off the shelf - see below in this chapter for details on querying
book = Book.objects.get(title="Moby Dick")
# Get the book's author - very simple
author = Book.author
# Get a set of the books the author has been credited on
books = author.book_set.all()
```

可以看到，这里从Author到Book的“反向关系”是通过Author.book_set属性来表示的（这是一个manager对象，本章稍后会讲到），它是由ORM自动添加的。你可以通过在ForeignKey里指定related_name参数来改变它的名字。在上面的例子里，我们如果把author定义成ForeignKey("Author", related_name="books")的话，最后就是访问author.books而不是author.book_set了。

> **注意**
>
> 对简单的对象层次来说，related_name不是必需的，但是在更复杂的关系里，比如当你有多个ForeignKey的时候就一定要指定了。这时，ORM需要你明确告诉它在ForeignKey的另一端如何区分两个不同的反向关系。如果你忘了的话，Django的数据库管理工具会抛出错误信息警告你！

多对多关系

外键通常用来定义一对多（或多对一）的关系——在上面的例子里，一本Book只能有一个Author，而一个Author却可以有很多本Book。不过有时候你需要更大的灵活性。比如之前我们假设每本Book只能有一个Author，那要是一本书同时有好几位作者的话怎么办，就好像本书？

这样的情况光在一段有“多”关系（Author有多本Book）是不够的，要两端都有才行（Book也可以有多个Author）。这就是多对多关系，由于SQL并没有定义这种关系，我们必须通过外键用它能理解的方式实现它。

Django为这种情况提供了第二种关系对象映射变量：ManyToManyField。语法上来讲，它和ForeignKey是一模一样。你在关系的一端定义它，把要关联的类传递进来，ORM就会自动为另一端生成使用这个关系必要的方法和属性（和前面看到的ForeignKey一样，通常就是生成一个_set manager对象）。不过由于ManyToManyField的特性，在哪一端定义它通常都没有关系，因为这个关系是对称的。

注意

如果你打算使用Django的admin，请注意一个多对多关系在admin里只有在定义关系的那一端才会显示关系对象。

注意

自引用的ManyToManyField（即一个给定模型为自己定义了一个ManyToManyField）默认就是对称的，因为它假设关系是相对的。然而，这也不是一定正确的，你可以在变量定义里通过指定symmetrical=False来改变这一行为。

根据这一“新发现”，我们必须处理一本书有多名作者的情况，因此修改例子如下：

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
```

ManyToManyField的用法和外键关系里“多”的那一端差不多：

```
# Pull a book off the shelf
book = Book.objects.get(title="Python Web Development Django")
# Get the books' authors
authors = Book.author_set.all()
# Get all the books the third author has worked on
books = authors[2].book_set.all()
```

ManyToManyField的秘密在于它在背后创建了一张新的表来满足这类关系的查询需要，而这张表用的则是SQL的外键，其中的每一行就代表了两个对象的一个关系，它同时包含了两端的外键。

这张查找表在Django ORM的使用过程中一般是隐藏起来的，不可以单独查询它，只能通过关系的某一端来进行查询。不过，你可以在ManyToManyField上指定一个特殊的选项，through，来指向一个显式的中间模型类。通过through你可以手动地管理这个中间类上的额外变量，同时还能继续方便地管理关系的两端。

下面的代码和之前ManyToManyField示例一模一样，除了它包含一个显式的中间表Authoring，它在关系上添加了一个collaboration_type变量，并通过through关键字指向这张表。

```
class Author(models.Model):
    name = models.CharField(max_length=100)
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author, through="Authoring")
class Authoring(models.Model):
    collaboration_type = models.CharField(max_length=100)
    book = models.ForeignKey(Book)
    author = models.ForeignKey(Author)
```

查询Author和Book的方法和之前完全一样，另外还能构造对“authoring”的查询。

```
# Get all essay compilation books involving Chun
chun_essay_compilations = Book.objects.filter(
    author__name__endswith='Chun',
    authoring__collaboration_type='essays'
)
```

如此，Django在创建关系的能力上就显得非常灵活了。

用一对一关系进行组合

除了常见的多对一和多对多关系类型外，关系数据库开发中有时候还需要第三种类型，一对一的关系。顾名思义，这个关系的意思就是两端都只有一个关联的对象。

Django实现这个OneToOneField概念的方式几乎和ForeignKey一样——它接受一个参数，即要关联的类（或者自引用时的“self”）。同样，它还接受一个可选参数related_name，这样就可以在两个相同的类里区分出多个这样的关系来。不同的地方在于，OneToOneField没有在反向关系里添加reverse manager，而只是增加了一个普通属性而已，因为关系的另一端一定只有一个对象。

这种关系类型最常用来支持对象组合或是拥有关系，所以相比现实世界，它更加面向对象一点。在Django直接支持模型继承（model inheritance）之前，OneToOneField主要是用来实现模型继承这类关系，而现在则是转向对这个特性的幕后支持。

限制关系

关于定义关系的最后一点，ForeignKey和ManyToManyField都可以指定一个limit_choices_to参数。这个参数接受一个字典，它的键值对是查询的关键字和值（下面会讲到这些关键字是什么）。这对指定你定义关系的可能值来说是一个非常强大的方法。

例如，下面这个版本的Book模型类只能和姓Smith的Authors类一起工作：

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class SmithBook(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
    })
```

> **注意**
>
> 你可以（有时候还最好是）在表单层上指定这个限制。参见第6章里对ModelChoiceField和ModelMultipleChoiceField的描述。

模型继承

在编写本书的时候，Django的ORM里一个相对比较新的特性就是模型继承（model inheritance）。两个模型类之间除了外键以及其他关系之外，还可以和普通的、非ORM的

Python类一样，通过从另一个模型继承来定义模型。（普通类的继承可以在第1章里找到。）

例如，上面的SmithBook类不仅可以定义成一个刚好和Book类一样有两个相同量的类，还可以显式地从Book类继承而来。这里的优点是很明显的——子类能通过添加或是重写变量来和父类区别开来，而不需要重复整个类的定义。

这里的Book示例太简单，显不出这个特性的优势。但是想象一下，如果是一个有一打甚至更多属性的模型，再加上一些复杂的方法，根据DRY原则，继承这种方式就突然变得很有竞争力了。不过别忘了你还是可以使用ForeignKey或者是OneToOneField的。到底使用哪种技术完全取决于你和你计划的模型设置。

Django目前支持两种不同的继承方式，每种都有自身的优点和缺点：抽象基础类（abstract base class）和多表继承（multi-table inheritance）。

抽象基础类

抽象基础类（abstract base class）这种方法简单来说就是“纯Python的”继承——它允许你重构Python模型的定义，这样变量和方法就可以从基类里继承下来。然而在数据库和查询层上并没有基类这个概念，子类在数据库的表其实还是复制了基类的变量。

这听起来有点违反DRY的原则了，但实际上在这个场景里是合理的，你可不希望为基类创建一张额外的数据表——特别是如果数据库是现有的，或是还有另一个应用程序在使用它的时候。另外这种无需隐含一个实际的对象层次就能表达类定义重构（refactoring of class definitions）的方法其实更整洁。

现在我们用抽象基础类来重新组织Book和SmithBook的模型层次。

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    num_pages = models.IntegerField()
    authors = models.ManyToManyField(Author)

    def __unicode__(self):
        return self.title

    class Meta:
        abstract = True

class SmithBook(Book):
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
    })
```

这里的关键是在Book的Meta嵌套类里的abstract = True设置——它指明了Book是一个抽象基础类，只是用来为它实际的模型子类提供属性而存在的。注意SmithBook只是重新定义了authors变量来提供它的limit_choice_to选项——因为它继承的不是常用的models.Model而是

Book，所以它的数据表里已经包含了title、genre和num_pages列，以及一个多对多到authors的查找表了。在Python类这一层上它还定义了一个__unicode__方法返回title变量的值，这和Book里一样。

换言之，当SmithBook在数据库里被创建，以及被用来创建对象，ORM查询等的时候，它的行为和下面的定义其实是完全一样的：

```
class SmithBook(models.Model):
    title = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    num_pages = models.IntegerField()
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
    })

    def __unicode__(self):
        return self.title
```

因为这一行为还延伸到了查询机制以及SmithBook实例的属性，所以下面的查询也是完全合法的：

```
smith_fiction_books = SmithBook.objects.filter(genre='Fiction')
```

不过其实我们的这个例子不是很适合用在抽象基础类里，通常你还是会需要创建普通Book类的。抽象基础类的当然是抽象的——它们不能被直接创建，而是和前面叙述的那样，最好是在模型定义层上提供DRY的支持。下面要讲到的多表继承（multi-table inheritance）则更适合我们这个特定的场景。

关于抽象基础类最后要说的是：在子类的嵌套类Meta会继承，或是和父类里Meta类合并起来（当然除了要把abstract选项重设为False，另外还有一些和数据库相关选项，比如db_name等）。

此外，如果基类用到了related_name参数来设置ForeignKey这样的关联变量的话，你需要用一些字符串格式化来避免子类的名字不会发生冲突。不要使用例如“related_employees”之类的常见字符串，而应该用%(class)s，比如这样“related_%(class)s”。（关于这种字符串替换的细节请参考第1章）这样，子类的名字就能正确替换，避免发生冲突。

多表继承

从定义上来说，多表继承（multi-table inheritance）和抽象基础类的差别不大。还是会用到Python的类继承，但是不再需要abstract = True这个Meta类选项了。在检查模型实例或是查询的时候，多表继承和我们前面看到的也都一样，子类会从父类继承所有属性和方法（除了Meta类里的一些例外，原因刚刚解释过了）。

这里主要的区别在于底层的机制。在这里，父类是拥有自己数据表的完整Django模型，可以正常地实例化，同时还能把自己的属性“借给”子类。其实，这是通过自动在子类和父类之间设置一个OneToOneField，以及幕后的一些小手段把两个对象连在一起来实现的，所以子类才能继承父类的属性。

所以换句话说，多表继承其实就是对普通“has-a”关系（或者说，对象组合）的一个方便的包装。因为Django希望尽量的Pythonic，所以如果你需要那个“被隐藏”起来的关系，其实可以通过OneToOneField显式的暴露出来，它会给出父类名字的小写形式并加上一个_ptr后缀。例如，在下面的例子里，SmithBook会得到一个指向其“父亲”Book实例的book_ptr属性。

下面的代码是用多表继承实现的Book和SmithBook示例：

```
class Author(models.Model):
    name = models.CharField(max_length=100)
class Book(models.Model):
    title = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    num_pages = models.IntegerField()
    authors = models.ManyToManyField(Author)

    def __unicode__(self):
        return self.title

class SmithBook(Book):
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
    })
```

前面提到过，这里唯一的不同就是没有Meta类的abstract选项。在一个空数据库和这个models.py文件上运行manage.py syncdb命令会创建三个主要的表（分别是Author、Book和SmithBook），而抽象基础类只会返回给我们两个表Author和SmithBook。

注意SmithBook实例得到的book_ptr属性会指向和它们组合的Book实例，而属于（或者一部分，取决于你怎么看待它）SmithBook的Book实例会得到一个smithbook（没有_ptr后缀）属性。

由于这种形式的继承允许父类拥有自己的实例，Meta的继承有可能会在关系的两端导致问题或冲突。所以，你需要重新定义绝大多数Meta选项，否则两个类都会分享它们（即使ordering和get_latest_by没有在子类里定义，它们也会被继承下来的）。这令遵循DRY原则变得稍微困难了一点，不过只要我们努力接近了就好了，哪有那么多十全十美的事情呢。

最后，我们希望你已经弄清楚了为什么这个方式比较适合我们的book模型，我们可以同时实例化普通的Book对象以及SmithBook对象。如果你要用模型继承来映射我们真实世界的关系，很多时候你都会发现多表继承比抽象基础类要好用得多。分辨究竟应该用哪一种（甚至两种都不行）是需要经验累积的。

Meta嵌套类

模型（model）里定义的变量（fields）和关系（relationships）提供了数据库的布局以及稍后查询模型时要用的变量名——经常你还需要添加__unicode__和get_absolute_url这样方法或是重写内置的save或delete方法。然而，模型定义里还有第三个方面，即用来告知Django关于这个模型的各种元数据信息的嵌套类Meta。

顾名思义，Meta类主要处理的是关于模型的各种元数据的使用和显示：比如在一个对象对

多个对象时，它的名字应该怎么显示；在查询数据表时默认的排序顺序是什么；数据表的名字是什么（如果你很在意这个的话）等。此外，多变量唯一性（multi-field uniqueness constraints）也是在Meta类里定义，因为这种限制没办法分开在每个单独的变量声明上定义。我们来给之前的Book类添加一点元数据。

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)

    class Meta:
        # Alphabetical order
        ordering = ['title']
```

这样就行了！Book类实在是太简单，大多数Meta嵌套类提供的选项都用不到，而且要不是我们真的关心默认排序方法的话，其实连这个Meta类都可以省略了。虽然Meta和Admin都很常用，但它们都是模型定义中可选的部分。Book这个例子太没劲了，现在我们来看一个复杂一点的例子。

```
class Person(models.Model):
    first = models.CharField(max_length=100)
    last = models.CharField(max_length=100)
    middle = models.CharField(max_length=100, blank=True)

    class Meta:
    # The proper way to order people, assuming a Last, First M. style of
    # display.
    ordering = ['last', 'first', 'middle']
    # Here we encode the fact that we can't have a person with a 100%
    # identical name. Of course, in real life, we could, but we'll pretend
    # this is an ideal world.
    unique_together = ['first', 'last', 'middle']
    # Django's default pluralization is simply to add 's' to the end: that
    # doesn't work here.
    verbose_name_plural = "people"
```

如同你在注释里看到的一样，如果没有Meta的帮助，对“人”这个概念建模还挺不容易的。在排序时必须同时考虑三个变量，要避开重名，还要在系统引用超过一个“人”的时候告诉它不要用“persons”这种古怪的名字。

如果想知道你还可以在Meta类里定义什么选项的细节的话，我们推荐你直接去看Django的官方文档。

Admin注册和选项

如果你正在用随Django一起发布的“admin”应用的话，一定也大量用到了admin的site对象和它们的register函数，可能还有ModelAdmin子类。这些子类允许你定义各种关于在admin应用里当你和模型交互的时候，模型应该如何使用的选项。

只需向admin注册你的模型类（并且激活Admin应用，参见第2章，），它就能为你提供关于这个模型基本的列表和表单；而挂接一个ModelAdmin子类则可以提供更多控制，例如允许你手动挑选列表视图里显示的变量，或是表单的布局等。

此外，你还可以通过创建Inline子类并在ModelAdmin子类里引用它们，来为相关的变量例如ForeignKey指定内联编辑选项（inline editing options）。这种膨胀类的方式初看起来有点奇怪，但它其实是一种十分灵活的方法，保证了任何给定的模型都可以用多种方法或是在多个admin站点里表示出来。另外，把模型层次扩展到内联编辑上就可以让你在需要的时候把内联表单放到多个“父”模型的页面里。

每个选项的具体定义我们就留给官方文档了（在第三部分里也有一些admin使用的例子），不过这里列出了两个ModelAdmin选项里最常见的主要类型。

- 列表格式化：list_display、list_display_links、list_filter等类似的选项允许你改变显示在列表视图里的变量（默认是在单独一列中模型实例的字符串表示），以及激活变量查找和过滤链接，这样就能迅速浏览你的信息了。
- 表单显示：fields，js、save_on_top等提供了很多灵活的方法来重写模型里默认表单表现形式，以及添加定制的JavaScript includes和CSS类，如果你想自己动手改变admin的外挂风格以便符合你整个网站的话，这些功能都是很有用的。

最后，如果你发现自己真的在大量使用这些选项，这其实代表了一个信号：你应该考虑舍弃admin转而编写你自己的管理表单。当然了，先读一下第11.1节，在开始动手之前看看Django的admin究竟有多少潜力可挖。

4.2 使用模型

到此我们已经解释了如何定义以及强化模型（model），接下来要详细讨论如何创建模型，通过模型查询数据库，最后是一些关于在底层支撑整个机制的原始SQL的要点。

用manage.py创建和更新数据库

第2章里提到过，随每个Django项目创建的manage.py脚本包含了操作数据库的功能。其中最常用的manage.py命令就是syncdb。不要被这个名字迷惑住，它并不是和有些用户想象的那样对整个数据库进行一次完整的同步。相反，它只是保证所有模型类都有对应的数据表，在必要时创建为模型创建新的表——但是不会去修改已经存在的数据表。

> **数据库“同步”**
>
> syncdb这种行为背后的原因在于Django的核心开发团队坚信生产数据（production data）绝不应该轻易交给一个自动化的过程。而且，通常认为只有当程序员足够了解SQL并能手动修改数据库时，才能修改数据库模式（schema）。我们对这一点也表示同意，在使用高级工具时多了解一些底层的技术总是好的。
>
> 不过一个自动化甚至半自动化的模型修改机制（例如Rails的各种变型）都经常可以加

快开发的进程。在编写本书的时候，已经有几个试图解决框架中这一不足的Django相关非核心项目正在开发中。

因此，如果你创建模型，运行syncdb将它载入数据库，然后改变这个模型时，syncdb不会试图去和数据库协调那些改动。它会希望程序员自己去手动修改，或是通过脚本修改，抑或者直接删掉数据表或整个数据库然后重新执行syncdb，这样就能得到最新的数据库模式（schema）了。就目前来讲，重要的是syncdb是把一个模型类转变成数据表最主要的方法。

除了syncdb，manage.py还提供了很多数据库相关的函数，syncdb实际上是构建在这些函数上来完成自己的工作的。表4.1给出了一些最常用的函数。这里有显示CREATE TABLE语句的sql和sqlall（sqlall还会执行一些初始数据的载入）；创建索引的sqlindexes；清空或删除数据表的sqlreset和sqlclear；以及执行应用程序自定义initial SQL语句的sqlcustom等。

表4.1 manage.py函数

manage.py函数	描 述
syncdb	创建所有应用程序所需的数据表
sql	显示CREATETABLE调用
sqlall	如同上面的sql一样从.sql文件中初始化数据载入语句
sqlindexes	显示对主键创建索引的调用
sqlclear	显示DROP TABLE调用
sqlreset	sqlclear和sql的组合（DROP加CREATE）
sqlcustom	显示指定.sql文件里的自定义SQL语句
loaddata	载入初始数据（和sqlcustom类似，但是这里没有原始SQL）
dumpdata	把现有数据库里的数据输出为JSON，XML等格式

和syncdb不同的是，这些sql*系列的命令并不更新数据库。相反，它们只是把对应的SQL语句打印出来，让程序员有机会验证（例如确保syncdb的行为和程序员期望的一致）或者把它们保存到单独的SQL脚本文件里去。

当然你也可以把这些命令的输出直接用管道重定向给数据库客户端来执行，这就好像将syncdb的步骤分解开来一样。你还可以通过重定向把两种方法组合到一个文件里，修改这个文件，随后把文件重定向给数据库执行（参见附录A来获得管道和重定向的细节）。

关于更多如何使用这些以及错综复杂的syncdb命令的信息，请参考第三部分里的应用示例，或是查阅Django官方文档。

查询语法

查询由模式生成的数据库需要使用两个不同但却相似的类：Manager和QuerySet。Manager对象总是附在模型类里，所以除非有特别指定，每个模型类都会展示一个objects属性，它构成了这个模型在数据库所有基本查询。Manager是从数据库获取信息的门户，它们包含了好几个方法来让你进行常见的查询。

- all：返回一个包含模式里所有数据库记录的QuerySet。
- filter：返回一个包含符合指定条件的模型记录的QuerySet。
- exclude：和filter相反——查找不符合条件的那些记录。
- get：获取单个符合条件的记录（没找到或是有超过一个结果都会抛出异常）。

当然，这里的内容有点超前了——我们还没介绍什么是QuerySet呢。可以简单地把QuerySet想象为模型类实例（或者说数据库行或记录）的列表，但它比列表要更强大一点。如果说Manager为生成查询提供了一个起点，那么QuerySet就是绝大多数查询发生的地方。

感谢Python的动态性、灵活性，以及所谓的“duck typing”特性，多功能的QuerySet对象提供了很多重要而强大的特性；它们可以查询数据库，可以包含数据，还可以组合在一起查询。

将QuerySet作为数据库查询

顾名思义，QuerySet可以被当作一个数据库查询的始发端。它可以被翻译成一个能在数据库上执行的SQL字符串。因为绝大多数SQL查询通常都是一组逻辑语句和参数匹配的集合，QuerySet完全可以在Python层上接受一个相同的版本。QuerySet接受动态的关键字参数然后转换成适合的SQL。我们用Book模型类来举个例子就清楚了。

```
from myproject.myapp.models import Book

books_about_trees = Book.objects.filter(title__contains="Tree")
```

这里关键字接受的是模型变量名（比如title）、双下画线和一个可选的说明比如contains，代表“大于”的gt，代表“大于等于”的gte，代表成员测试的in等。每一个都直接（或接近）映射到SQL操作符和关键字上去。官方文档上有全部关于这些操作符的细节。

回到这个例子上来，Book.objects.filter是一个Manager方法，前面解释过了，Manager方法总是返回QuerySet对象。在这里，我们要求Book的默认manager返回所有标题里含有“Tree”字样的书，并且将返回的结果QuerySet存放在一个变量里。这个QuerySet所代表的SQL查询是这样的：

```
SELECT * FROM myapp_book WHERE title LIKE "%Tree%";
```

组合查询也是可以的，以之前的Person模型来举例：

```
from myproject.myapp.models import Person

john_does = Person.objects.filter(last="Doe", first="John")
```

而这里对应的SQL就是：

```
SELECT * FROM myapp_person WHERE last = "Doe" AND first = "John";
```

使用其他Manager的方法也是返回相似的结果，例如all：

```
everyone = Person.objects.all()
```

所对应的SQL显然就是：

```
SELECT * FROM myapp_person;
```

请注意在可选的Meta模型嵌套类里定义的那些各种和查询相关的选项都会影响到所生成的SQL，比如，ordering会被转换成ORDER BY。稍后会看到，QuerySet的其他方法以及其他能力也能改变最终在数据库上执行的SQL。

最后，如果你会使用SQL和理解各种查询机制背后所隐含的意义的话（无论是结果集还是查询期），那么在构建ORM查询时会更加得心应手，能比不懂SQL的人构造出更快、目的更明确的查询来。此外，Django正打算将来允许用户更容易地访问QuerySet对象并调整所生成的SQL——这样你就能获得更多控制权了。

将QuerySet作为容器

QuerySet很像是一个列表。它实现了一部分列表的接口因此你可以执行迭代（for record in queryset:），索引（queryset[0]），切片（queryset[:5]），以及获取长度（len(queryset)）等操作。所以如果你熟悉Python的列表、元组或是迭代器的话，访问QuerySet里包含的模型对象也是一样的。而且这些操作都颇为智能，比如切片和索引操作都会被自动传换成SQL里的LIMIT以及OFFSET关键字。

有时候，你会发现需要QuerySet做的事情不可能，至少也是不适合用Django ORM现有的功能完成。这时只需用list函数把QuerySet转换成一个列表就行了，这是一个包含了结果集的真正的列表。虽然这样有时候是必要的，也很好用（比如需要在Python层面上排序时），但是请注意如果QuerySet里的结果包含很多对象的话，这么做会导致内存或数据库的巨大负担！

Django已经尽可能地为ORM提供了很多强大的功能，所以如果你觉得需要把QuerySet转换成列表的时候，最好先停下来浏览一下本书或是官方文档，或者搜索一下Django邮件列表的存档。很可能就会找到不需要把整个QuerySet装到内存里就能解决问题的办法了。

组合QuerySet查询

QuerySet是懒惰的。只有在必要时它才会去执行数据库查询，比如当被转换成列表或是被前面几节里提到的访问方式访问的时候。这种行为是QuerySet里最强大的特性之一。它们不必是一个单独的、一次性的查询，而可以是复杂或是嵌套查询的组合。这是因为QuerySet暴露了很多和Manager一样的方法，例如filter和exclude等。和Manager一样，这些方法会返回新的QuerySet对象——但这时它们进一步被限制在父QuerySet自身的参数范围里了。举个例子会容易理解一点。

```
from myproject.myapp.models import Person

doe_family = Person.objects.filter(last="Doe")
john_does = doe_family.filter(first="John")
john_quincy_does = john_does.filter(middle="Quincy")
```

根据重要的程度，我们一步步地缩小查询结果的范围，直到最后只剩下一个结果（反正不会很多，取决于数据库里有多少个叫John Quincy Doe的人）。因为这些都是Python代码，所以可直接把它们写成一行。

```
Person.objects.filter(last="Doe").filter(first="John").filter(middle="Quincy")
```

机敏的读者一定觉得有了Person.objects.filter，没有什么事情是做不到的了。但是，如果之前的john_does的QuerySet不是得自函数调用而是从某个数据结构里取得的话怎么办？这时我们根本不知道我们在处理的是什么查询内容——当然我们也不一定需要知道。

假设给Book模型添加一个due_date变量，负责显示所有逾期未还的书（即，还书日比今天早的书）。我们可能得到的是图书馆里所有书，或是所有小说书，又或者是所有某个人归还的书等——即任何形式的集合都是有可能的。而我们可以将这样一个集合缩小到只显示我们感兴趣的书，比如逾期不还的书。

```
from myproject.myapp.models import Book
from datetime import datetime
from somewhere import some_function_returning_a_queryset

book_queryset = some_function_returning_a_queryset()
today = datetime.now()
# __lt turns into a less-than operator (<) in SQL
overdue_books = book_queryset.filter(due_date__lt=today)
```

除了这种形式的过滤，任何复杂的逻辑都需要使用QuerySet组合，例如所有作者名为Smith的非小说类书籍。

```
nonfiction_smithBook.objects.filter(author__last="Smith").exclude(genre="Fiction")
```

虽然用查询选项里的否定操作也能达到同样的效果，比如（Django ORM曾经支持过的）as__genre__neq等，但是把逻辑放到额外的QuerySet方法里能让结构变得更清楚。还有这种把查询分成多个独立步骤的方式也更加易读。

查询结果排序

你应该注意到了QuerySet里有一些Manager对象中没有的方法，因为它们的作用的是修改查询的结果而不是生成新的查询。其中最常用的就是order_by，它能改变QuerySet默认的排序方法。例如，Person类通常是按照姓氏排序的，我们可以把某个单独QuerySet改成按名字排序，像这样：

```
from myproject.myapp.models import Person

all_sorted_first = Person.objects.all().order_by('first')
```

QuerySet结果的其他部分还是一样的，除了背后SQL层上的ORDER BY子句改成了按我们的要求进行了更新。这就是说我们可以在上面加诸更多东西来组成更复杂的查询，例如查找按名字排序里的前五个人。

```
all_sorted_first_five = Person.objects.all().order_by('first')[:5]
```

甚至可以用前面见过的双下画线语法来排序模型关系。假设Person有一个ForeignKey包含了一个Address，Address里有一个state变量，我们希望按照他所在的州、然后是姓氏，对这些人进行排序。像这样：

```
sorted_by_state = Person.objects.all().order_by('address__state', 'last')
```

最后，你可用通过在指定字符串上加一个减号来完成倒序排列，例如order_by('-last')。甚至还可以调用QuerySet的reverse方法将整个QuerySet倒过来（例如QuerySet是从代码里传递过来，不能像前面那样直接控制order_by调用的话）。

其他改变查询的方法

除了排序，还有一些值得注意的只属于QuerySet的方法，例如distinct，它会通过SQL的SELECT DISTINCT移除QuerySet里所有重复的元素。还有values，它接受一列变量名（包括关系对象上的变量）并且返回一个QuerySet的子类ValueQuerySet，这个子类里只以字典形式保存了一列的刚刚要求的变量，而不是普通的模型类。values还有一个类似的方法values_list，它返回的是一列元组而非字典。这里是values和values_list的几个例子。

```
>>> from myapp.models import Person
>>> Person.objects.values('first')
[{'first' : u'John'}, {'first': u'Jane'}]
>>> Person.objects.values_list('last')
[(u'Doe',), (u'Doe',)]
```

另一个好用但是容易被忽略的是select_related方法，它有时能帮助解决ORM里一个常见的问题，即对概念上很简单的操作却在背后执行了很多无谓的查询。例如，假设你要查询大量的Person对象然后显示关于他们的Address对象信息（想想上一节里的场景），数据库就会先去查询一个Person的列表，然后去对每个Address做一次查询。要是你的列表里有成百上千的Person的话，这个查询量就实在太大了！

为了避免这个问题，select_related会自动在数据库上执行join操作来把相关的对象组合起来，所以你只需要执行一个更大的查询就可以了——数据库在少量的大型查询上比大量的小规模查询效率要高的多。不过请注意select_related方法在null=True时不会去查找关系，所以在设计模型的时候就要考虑到这一点性能。

最后还请注意，你还可以用depth参数来控制select_related在关系链上能走到“多远”，以防止在对象层次上深入太多而生成太大的查询。如果对象层次比较宽广，你甚至可以通过传入变量名来选择其中特定的几个关系。

下面这个例子展示了如何使用select_related来执行一个简单的join操作，把Person及其对应的地址合并起来查询并忽略其他在Person或Address里定义的ForeignKey：

```
Person.objects.all().select_related('address', depth=1)
```

例子是很简单，但是意思在这里了。select_related这类方法只有在你需要更多查询引擎默认行为的时候才显得有用。如果你没有过中型或是大型网站的经验，这些方法看起来还不是很有用，但是当你的网站开发完成，需要考虑性能的时候，它们绝对是你不可或缺的好帮手。

所有这些函数（包括order_by和reverse）的细节都可以在Django官方文档里找到。

用Q和~Q组合查询关键字

QuerySet可以通过一个叫Q的关键字参数封装类进一步参数化，它允许你使用更复杂的逻辑，例如用 & 和 | 操作符来组成AND和OR关系（虽然它们和Python里对应的操作符and和or，

或是位操作符 & 和 | 很像，但是千万不要弄混了）。其结果Q对象可以用来作为filter或是exclude方法的关键字参数，例如：

```
from myproject.myapp.models import Person
from django.db.models import Q

specific_does = Person.objects.filter(last="Doe").exclude(
    Q(first="John") | Q(middle="Quincy")
)
```

虽然这个例子有点做作（大概不太会有人关心一个特定的名字或是中间名），不过这里主要是为了展示Q的用法。

和QuerySet一样，Q对象也可以组合在一起使用，通过 & 和 | 操作符，你可以返回和操作数对应的新的Q对象。比如，你可以像这样用循环来组装出一个巨大的查询。

```
first_names = ["John", "Jane", "Jeremy", "Julia"]

first_name_keywords = Q() # Empty "query" to build on
for name in first_names:
    first_name_keywords = first_name_keywords | Q(first=name)

specific_does = Person.objects.filter(last="Doe").filter(first_name_keywords)
```

这里，我们创建了一个for循环用 | 操作符把所有列表里生成的Q对象组合起来。虽然这个例子举的不是很好（这种情况其实用__in查询操作符就可以了），但是它展示了这种通过编程的方式将Q对象组合起来的潜在能力。

> **注意**
>
> 如果上面的例子要是用上Python函数式编程工具的话（比如列表推导式，内置的reduce方法，以及operator模块），还能更简洁一点。operator模块提供了和操作符等价的函数，例如和 | 对应的or_，还有和 & 对应的and_。这样那三行for循环就可以重写成这个样子：reduce(or_, [Q(first=name) for name in first_names])。不要忘了，Django其实“就是Python”，这类手法可以用在框架里的任何一部分里。

最后，你可以对Q对象使用一元操作符~来取反。虽然QuerySet的exclude方法可能是更直观一点，不过~Q在逻辑很复杂的时候就能显示出优势来。比如在下面这个例子里，只用一行就能获取所有姓Doe，名John Smith，但是不叫John W.Smith的人。

```
Person.objects.filter(Q(last="Doe") |
    (Q(last="Smith") & Q(first="John") & ~Q(middle__startswith="W"))
)
```

要是这里用exclude(middle_startswith="W")的话就不对了（它会排除所有姓Doe，中间名是“W”的人，这就不是我们要的结果了），但是用~Q就能正确表达出这个意图。

用Extra调整SQL

关于Django查询机制的最后一点（以及引导到下一节，它现在做不到的内容），我们来看看QuerySet的extra方法。这是一个多功能的方法，可以用来修改QuerySet生成的原始SQL的各个部分，它接受四个关键字参数，如表4.2。注意为了更好地展示概念，这一节的例子里会用到一些没有在之前模型里定义的属性。

表4.2 extra接受的参数

extra参数	描 述
select	修改SELECT语句
where	提供额外的WHERE子句
tables	提供额外的表
params	安全的替换动态参数

select参数接受一个映射到SQL字符串的标识符字典，让你可以根据在SQL SELECT子句里的选择为生成的模型实例添加定制的属性。这在你希望在从数据库获取的信息之外再添加一些的时候，以及把这些改动限制在代码中的一部分的时候（而不是应用在模型方法上，因为这些方法会在所有的地方执行）就很方便。而且，有些操作在数据里执行就是要比在Python里快，所以也可以用来做优化。

这个例子是用select来添加一个简单的数据库逻辑的属性：

```
from myproject.myapp.models import Person


# SELECT first, last, age, (age > 18) AS is_adult FROM myapp_person;
the_folks = Person.objects.all().extra(select={'is_adult': "age > 18"})

for person in the_folks:
    if person.is_adult:
        print "%s %s is an adult because they are %d years old." % (person.first,
            person.last, person.age)
```

where参数接受一个字符串列表作为参数，列表里包含的是原始的SQL WHERE子句，直接被加入到最终的SQL查询as-is里去（或者说几乎是as-is，参见下面的params）。where最适合在无法用属性相关的关键字参数如__gt或__icontains组成正确查询的情况下使用。在下面的例子里，我们用了同一个SQL结构来搜索和返回一个用PostgreSQL风格的连接符||连接的字符串：

```
# SELECT first, last, (first||last) AS username FROM myapp_person WHERE
# first||last ILIKE 'jeffreyf%';
matches = Person.objects.all().extra(select={'username': "first||last"},
    where=["first||last ILIKE 'jeffreyf%'"])
```

extra的参数里最简单的可能就是tables了，它允许你指定一个包含额外数据表名字的列表。这些名字随后会被插入到查询的FROM子句里，一般是用JOIN语句串联起来。在默认情况下，Django会以appname_modelname这样的格式命名你的数据表。

这里是一个tables的例子，简单起见，它用的模型和别的不太一样（回到了Book类这个例子上，还多加了一个author_last的属性）。

```
from myproject.myapp.models import Book

# SELECT * FROM myapp_book, myapp_person WHERE last = author_last
joined = Book.objects.all().extra(tables=["myapp_person"], where=["last =
author_last"])
```

最后一个参数就是params。在高级程序语言里进行数据库查询的“最佳实践”之一就是正确地转义或插入动态参数。初学者常犯的一个错误就是简单地用字符串连接来把他们的变量插入到SQL查询里去，但是这样做会给系统带来潜在的安全漏洞和bug。

所以在使用extra时，应该使用params关键字（它接受一个要替换值的列表）来替换where字符串里的%s字符串占位符，像这样：

```
from myproject.myapp.models import Person
from somewhere import unknown_input

# Incorrect: will "work", but is open to SQL injection attacks and related problems.
# Note that the '%s' is being replaced through normal Python string interpolation.
matches = Person.objects.all().extra(where=["first = '%s'" % unknown_input()])

# Correct: will escape quotes and other special characters, depending on
# the database backend. Note that the '%s' is not replaced with normal string
# interpolation but is to be filled in with the 'params' argument.
matches = Person.objects.all().extra(where=["first = '%s'"],
    params=[unknown_input()])
```

利用Django没有提供的SQL特性

最后关于Django模型/查询框架的一点就是，ORM并不能完整地覆盖所有可能性。几乎没有什么ORM敢自称能100%地替换常规的数据库接口。Django也不例外，即便它的工程师们已经并且一直在努力提升它的灵活性。有时候，特别是对有丰富的关系数据库经验的人来说，避开ORM也是相当有必要的。下面的几节就是关于这个话题的内容。

定义模式（schema）和定制initial SQL

除了标准的表和列，大多数关系型数据库还会提供一些额外的特性，例如视图或统计表（view），触发器（trigger），定义在数据行被删除或更新时“联级”（cascade）行为的能力，甚至在SQL层上自定义函数或数据类型等。在本书编写的时候，Django ORM（和几乎所有ORM）在很大程度上都无视了这些东西，但这不是说你就不能用它们了。

Django的模型定义框架里一个才加入不久的特性就是定义initial SQL文件的能力，它们必须存放在应用程序里的sql子目录下并以.sql结尾，比如myproject/myapp/sql/triggers.sql。任何这样的文件都会在你运行manage.py和SQL相关的命令如reset或syncdb，以及被包括到sqlall或sqlreset输出里的时候自动执行。这个特性还有一个自己的manage.py命令，sqlcustom，它（和

其他sql*系的命令一样）会打印出它找到的自定义SQL。

通过使用initial SQL，你可以将模式定义命令存放在Django项目里，并且你知道当使用Django的工具来创建或者重新创建数据库时，它们一定会被包括进来。下面这些功能都可以通过使用这个特性来完成。

- 视图：因为SQL的视图实际上就等于是只读表，所以你可以用创建Model定义来映射这些表的布局这种方式来支持它们，然后你就可以通过普通的Django查询API来访问它们了。注意要小心不要执行任何试图修改这样一个模型的manage.py SQL命令，否则就会发生问题。因为SQL库访问视图时，任何对视图表的写操作都会导致错误。
- 触发器和联级：它们都能和普通ORM方法生成的插入或更新操作一起正常使用，而且根据数据库的不同，还可以定义在initial SQL文件里。（如果联级限制不能创建的话，也可以用通过manage.py sqlall输出手动添加。）
- 自定义函数和数据类型：这些都可以在initial SQL文件里定义，但是要在ORM里引用的话需要使用QuerySet.extra。

Fixtures：载入和导出数据

虽然技术上说起来这不算是SQL本身的内容，但我们还是决定在这一节看一下这个在ORM之外和数据库相关的Django特性：fixtures。在第2章里我们已经简单地接触过它了，fixtures是保存在纯文本文件里数据库数据集合的名字，通常指的不是原始的SQL dump，而是一种数据库无关的（通常是可读的）表现形式，例如XML、YAML，或是JSON。

fixtures最常见的用法就是在数据库新建或是重建的时候，作为初始数据加载进来，例如用来给用户输入的数据分类用的“预装”的数据表，或是在开发期间载入的测试数据。Django支持这个功能的方式和前面介绍的initial SQL很相似，每个Django应用都有一个fixtures子目录，里面包含了一个initial_data.json文件（可以是.xml、.yaml，或是其他序列化格式）。每当运行数据库创建/重设命令，比如manage.py syncdb或reset的时候，Django的序列化模块（参见第9章）就会读取这些文件并用其内容创建数据库对象。

这里是一个给Person模型类准备的例子，一个简单的JSON fixture文件：

```
[
    {
        "pk": "1",
        "model": "myapp.person",
        "fields": {
            "first": "John",
            "middle": "Q",
            "last": "Doe"
        }
    },
    {
        "pk": "2",
        "model": "myapp.person",
        "fields": {
            "first": "Jane",
```

```
            "middle": "N",
            "last": "Doe"
        }
    }
]
```

导入数据库后的输出为：

```
user@example:/opt/code/myproject $ ./manage.py syncdb
Installing json fixture 'initial_data' from '/opt/code/myproject/myapp/fixtures'.
Installed 2 object(s) from 1 fixture(s)
```

除了初始化数据，fixtures通常还被用作是一个比数据库SQL dump工具更中立的（虽然有时比较低效或不太专用）数据库dump格式——例如，你可以把一个Django应用的数据从PostgreSQL里导出来，随后载入到MySQL数据里，如果没有一个中间转换的fixtures步骤的话，这类任务还真不容易。它们的命令是manage.py dumpdata和loaddata。

使用dumpdata和loaddata的时候，fixtures文件的位置和文件名比用作初始化数据的时候灵活一点。文件名可以任取（只要扩展名是可支持的格式就行了），文件的位置可以是fixtures子目录，任何FIXTURES_DIRS设置里的目录，甚至可以由loaddata或dumpdata显式的提供目录。例如，我们可以把之前导入的两个Person对象导出来（使用indent选项可以让输出更易读）。

```
user@example:/opt/code/myproject $ ./manage.py dumpdata  -indent=4 myapp >
➥/tmp/myapp.json
user@example:/opt/code/myproject $ cat /tmp/myapp.json
[
    {
        "pk": 1,
        "model": "testapp.person",
        "fields": {
            "middle": "Q",
            "last": "Doe",
            "first": "John"
        }
    },
    {
        "pk": 2,
        "model": "testapp.person",
        "fields": {
            "middle": "N",
            "last": "Doe",
            "first": "Jane"
        }
    }
]
```

就像你看到的一样，fixtures能让你用比原始SQL简单一点的格式处理数据。

自定义SQL查询

如果ORM（包括extra带来的灵活性）怎么样都无法满足你查询的需要的话，你总是可以回

到底层的数据库接口上执行自定义SQL语句的。Django的ORM也是使用这些模块来和数据库交互的。根据Django数据库设置的不同，这些模块也会随之变化，通常它们都是符合Python DB-API标准的。只需要导入django.db里定义的connection对象，然后就可以获取数据库游标进行查询了。

```
from django.db import connection

cursor = connection.cursor()
cursor.execute("SELECT first, last FROM myapp_person WHERE last='Doe'")
doe_rows = cursor.fetchall()
for row in doe_rows:
    print "%s %s" % (row[0], row[1])
```

> **注意**
>
> 要知道更多关于这些模块提供的语法和方法调用的细节，请参阅Python DB-API文档，《Core Python Programming》中的“数据库”一章，或是数据库适配器文档（见withdjango.com）。

4.3 总结

本章覆盖了很多内容，我们希望你能从中得到一些有用的想法可以（也应该）带到你自己的数据模型里去。你了解了使用ORM的原因，学习了如何定义简单的Django模型，以及如何定义它们之间复杂的关系，还看到了Django如何使用特殊的嵌套类来指定模型的元数据。此外，希望你也同意QuerySet类作为从数据库获取数据手段的强大与灵活，并且了解了如何在ORM之外操作Django应用程序的数据。

在下面的两章，第5章和第6章里，你会学习到如何在一个Web应用里使用你的模型，比如在控制器逻辑（视图）里设置查询，然后在模板里显示它们。

第5章 URL、HTTP机制和视图

在上一章里，我们学习了如何定义支撑几乎所有Web应用的数据模型（model）。在这一章里，我们要向你展示如何通过Django的模板语言和表单显示这些模型。但是，一个Web框架只有这两个方面是不够的，你还需要控制器逻辑来决定哪个模板渲染什么数据，以及URL分派机制来决定对于给定的URL应该执行什么逻辑。

这一章消息介绍了Django是如何实现在第3章里介绍的HTTP请求-响应架构，然后解释了构成控制器逻辑的Python函数，以及一些协助常见任务的内置帮助函数。

5.1 URL

将一个请求的URL和结果的响应联系起来的机制就是任何Web开发框架的关键所在。Django使用的是一种简单而强大的机制，让你可以通过正则表达式配置映射到Python的视图方法上，并且通过相互包含把它们链接在一起。这样的系统不但使用方便，且灵活多变。

URLconf介绍

刚刚提到的映射存放在叫做URLconf的Python文件里。这些文件都必须暴露出一个urlpattern对象，这个对象则应该是Django定义的patterns函数的结果。这个被调用的patterns函数由以下两点组成：

- 一个打头的前缀字符串，可以为空。
- 一个或多个由正则表达式（regex）字符串匹配一个或一组URL组成的Python元组；一个视图函数对象或字符串；有时还可以带上一个视图函数的字典参数。

举个例子就清楚了，我们用在第2章里的Blog应用程序中的URLconf为例：

```
from django.conf.urls.defaults import *

urlpatterns = patterns('myproject.myapp.views',
    (r'^$', 'index'),
    (r'^archives/(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})/$', 'archive'),
)
```

显然，这里的正则表达式（参见第1章）才是重中之重。除了缺少打头的斜杠外（因为它到处都是，故而省略了），最先要查找的就是分别代表字符串起始和结束的正则表达式字符，^和$。

实际上，^的作用主要是消除匹配的二义性。URL /foo/bar/ 和 /bar/是不同的，但是正则表达式r'bar/'却都能匹配它们，而r'^bar/'则更明确的多，它只会匹配后面那个字符串。

使用$字符通常也是基于相同的原因。它确保正则表达式只会匹配URL的最后而不是中间。

但是如果某个URL项要包含其他URLconf文件的话，就不需要这个 $ 了，因为一个要包含其他URL的URL不是最终的URL，只是其中的一部分而已。

> **注意**
>
> 如果你注意一下前面例子的第一个元组，就会发现它只包含了r'^$'。这代表它要匹配的是网站的根URL，/。前面说过，Django会去掉打头的斜杠，所以在字符串的开头（^）和结尾（$）之间就只剩下一个空字符串了。在Django项目里你会经常用到这个来定义首页。

这些正则表达式里另一个要注意的地方就是它们可以使用一种叫符号组（symbolic groups）的语法来捕捉部分URL的变化。这个特性提供了定义动态URL必要的能力。在上面的例子中，我们有一个blog存档的部分，可以通过日期来定位一个帖子。被URL访问的基本信息（blog帖子）并没有变化，只是通过日期来访问它而已。就像刚才看到的那样，这里捕捉的值会被传递到特定的视图函数里去，然后函数会在数据库查询里或是任何合适的地方使用它们。

最后，定义完正则表达式以后，接下来要注意的是它所链接的函数以及可选的关键字参数（这里以字典形式存放）。patterns的第一个参数如果非空的话就会被加到函数字符串之前。拿刚刚的例子来说，前缀字符串是“myproject.myapp.views”，所以“index”和“archive”最终完整的Python模块路径名分别就是“myproject.myapp.views.index”和“myproject.myapp.views.archive”。

用url替换元组

url方法是最近才加入到Django的URL分派机制里的，作用是要在替换前面例子里的那些元组的同时尽量保持结构的一致性。它也接受三个“参数”（一个正则表达式、一个视图字符串/函数和一个可选字典参数）并且添加了另一个可选的命名参数：name。name就是一个字符串，它必须在所有的URL里保持唯一，你可以在别的地方通过它引用这个特定的URL。

我们来用url重写刚才的例子。

```
from django.conf.urls.defaults import *

urlpatterns = patterns('myproject.myapp.views',
    url(r'^$', 'index', name='index'),
    url(r'^archives/(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})/$', 'archive',
        name='archive'),
)
```

你可以看到，这就是对旧有的元组语法的一个简单的替换。因为它是一个实际的函数而不仅仅是一个元组，它迫使以前的语法变成一个约定而已。前两个参数是必需的且没有名字，字典参数现在是一个有名字的可选参数kwargs，以及一个新的可选命名参数name。

> **注意**
>
> kwargs和name都是命名参数（named argument）而不是位置参数（positional argument）因为它们都是可选的。你可以一个都不指定，指定其中一个，或者两个都指定。位置参数（或使用元组）会让设置变得困难许多。

在基于元组的语法后面介绍url方法是因为它比较新，即便是你读到这里的时候，外界肯定还是使用元组多过使用url来设置Django URLconf。不过，我们还是强烈推荐你在代码里使用url方法。在本书余下的部分里，我们会努力用它来做出一个好榜样，因为它比元组方式更强大也更灵活。

最后，要知道更多关于name参数的信息以及如果通过它从代码的其他部分逆向引用URL的细节，请参见第三部分中的应用示例。

使用多个patterns对象

Django程序员常用的一个技巧就是重构URLconf，把URL打散到多个patterns调用里，至少是对那些条目数目很大的文件来说。可以这么做的原因是patterns的返回类型是一个Django的内部对象类型，它可以被当作列表或是其他容器类型一样附加元素。这样就可以很方便地将多个这样的对象连接起来，所以一般都会按照前缀字符串来将它们分隔开来。这里是一个半抽象的例子，它表示了一个连接有多个应用的顶层URL。

```
from django.conf.urls.defaults import *

urlpatterns = patterns('myproject.blog.views',
    url(r'^$', 'index'),
    url(r'^blog/new/$', 'new_post'),
    url(r'^blog/topics/(?P<topic_name>\w+)/new/$', 'new_post'),
)

urlpatterns += patterns('myproject.guestbook.views',
    url(r'^guestbook/$', 'index'),
    url(r'^guestbook/add/$', 'new_entry'),
)

urlpatterns += patterns('myproject.catalog.views',
    url(r'^catalog/$', 'index'),
)
```

注意这里第二和第三个patterns调用里+=操作符的使用。在文件的最后，urlpatterns包含了一共6个URL定义，而且归功于这些不同的前缀参数，每个URL都有自己不同的映射。当然，聪明的读者一定注意到了这还不是重构的尽头。URL定义里的“blog”，“guestbook”，和“catalog”部分还是有点重复。所以下面我们来看看如何通过包含其他URLconf来进一步的精简它。

用include来包含其他URL文件

上一节里看到的重构思想可以进步深入下去，将URLconf文件分成多个这样的文件。这在含有多个应用的项目里很常见，比如可以有一个“基础的”应用程序定义首页或者其他用于整个站点的特性，如认证等。那么这个基础应用程序的URLconf会定义一些子小节让其他应用程序来填充，并且用一个特殊的include函数来将URL分派传递到其他应用程序里去，这里是之前例子的更新版：

```
## urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('myproject.blog.views',
    url(r'^$', 'index'),
    url(r'^blog/', include('myproject.blog.urls'),
)

urlpatterns += patterns('',
    url(r'^guestbook/', include('myproject.guestbook.urls')),
)

urlpatterns += patterns('',
    url(r'^catalog/', include('myproject.catalog.urls')),
)

## blog/urls.py

urlpatterns = patterns('myproject.blog.views',
    url(r'^new/$', 'new_post'),
    url(r'^topics/(?P<topic_name>\w+)/new/$', 'new_post'),
)

## guestbook/urls.py

urlpatterns += patterns('myproject.guestbook.views',
    url(r'^$', 'index'),
    url(r'^add/$', 'new_entry'),
)

## catalog/urls.py

urlpatterns += patterns('myproject.catalog.views',
    url(r'^$', 'index'),
)
```

这个例子比前面的版本要长一点，但是你可以看到对一个现实世界中每部分都包含数个URL的网站来说这种写法好处多多。在其中的每个部分，我们都避免了在URL定义里重复输入“blog”、“guestbook”和“catalog”。特别是现在我们把这个多应用网站里的绝大多数URL都交给了各自的子应用去处理，除了在blog应用下的首页（当然你也可以为它创建一个base或者类似的应用——这完全都是由你安排的）。

URLconf包含（URL including）即使对单个应用也是有价值的——因为使用多个应用还是一个应用根本就没有硬性规定，所以完全可以在一个Django应用里包含几百个URL。当然大多数程序员在这种情况下会迅速开始为它们组织模块，并采用URLconf包含来支持它们。通常来说怎么组织站点是由你决定的，而URLconf机制已经被设置的尽可能的灵活，其中包含（includes）又占了很大的比重。

函数对象 v.s. 函数名字符串

在这一节里，我们已经看到了URL所链接到视图函数里的Python模块路径可以用字符串来指定。不过这不是唯一的方法，Django还允许你传入一个可调用的对象（callable object），例如：

```
from django.conf.urls.defaults import *
from myproject.myapp import views

urlpatterns = patterns('', # Don't need the prefix anymore
    url(r'^$', views.index),
    url(r'^blog/', include('myproject.blog.urls')),
)
```

这样一来，它的功能就非常强大了，比如可以使用装饰器函数（decorator function）来包裹通用视图（generic view），甚至创建你自己的可调用对象将更复杂的工作代理给其他视图。第11章会讨论更多关于装饰器以及使用可调用视图的一些其他技巧。

> **注意**
>
> 有时候在URLconf里把from myproject.myapp.views import * 覆盖掉来用可调用视图的这种想法很有诱惑力，但是这在混合多个视图模块的时候有可能会产生问题——比如两个不同的视图文件都定义了它们自己的index视图。所以比较聪明的做法就是学习上面的例子那样，把每个视图模块导入为它自己的对象（必要时可以使用from x import y as z这种语法），这样我们可以保持本地名字空间不被污染。

5.2 HTTP建模：请求、响应和中间件

你已经了解了如何设置URL定义以及如何将URL和视图函数联系起来，现在我们来看看这些视图函数的周围都有些什么。第3章里说到，Django把HTTP归结为相对简单的Python请求对象和响应对象。加上URL转发和视图函数，一个请求在你的Web应用程序的流程是这样的：

- Web服务器接到一个HTTP请求。
- Django把Web服务器传过来的请求转换成一个请求对象。
- Django在URLconf里查找正确的视图函数。
- 调用这个视图函数，参数为请求对象以及任何捕捉到的URL参数。
- 然后视图会创建并返回一个响应对象。
- Django将这个响应对象转换成Web服务器可以理解的格式。
- Web服务器将响应发送给客户端。

首先我们看一下请求和响应对象以及它们的组件，然后深入Django中间件，它提供了进入上述过程中多个步骤的接口。之后，下一节的主要内容就是讲解关于视图本身你所需要了解的知识。

请求对象

设置完URLconf之后，你就需要定义URL所对应的行为。视图方法的细节稍后会详细介绍，现在我们让你看看视图处理的HTTP请求和响应对象是什么样子的。所有视图函数都接受一个“请求”参数HttpRequest对象，它是一组经过良好封装的属性，代表了从Web服务器传递进来的原始HTTP请求。

GET和POST字典

Web程序员最常用到的请求数据就是GET和POST数据结构，它们是HttpRequest对象的属性（即request.GET和request.POST）并以Python字典的形式出现。虽然它们结构相同，但是填充的方法却很不一样，这里面的区别比你乍看之下要重要的多。它们一起为参数化Web请求提供了一个灵活的方式。

> **注意**
>
> 虽然HttpRequest的GET和POST属性看起来和Python的内置dict非常的像，不过其实它们是Django专用的dict子类，QueryDict的实例，这个类型是设计用来模仿在底层HTTP CGI标准里这些数据结构的行为的。所有的键-值对里的值都是列表，即使对只有一个值的键也是一样，以便在HTTP服务器万一真的返回多个值的时候能正确处理它们。不过方便起见一般QueryDict都是和普通字典一样返回一个单值。如果你对多值都兴趣的话，可以使用QueryDict的方法，比如getlist来获取它们。

GET的参数是作为URL字符串的一部分传递进来的，但是并不属于URL本身，因为它们没有定义另一个资源（或视图），只是改变它们所对应的资源的行为罢了。例如URL /userlist/ 指向了一个列出网站社区上用户的页面，如果用户希望不要一次列出那么大的一个列表，他可以通过GET变量告诉服务器所需的页码：/userlist/?page=2。这样，虽然被访问的还是这个视图，但是程序员可以在GET字典里查找一个page的键值对，然后返回正确的页面，像这样：

```
def userlist(request):
    return paginated_userlist_page(page=request.GET['page'])
```

注意这里的request.GET（包括请求对象里其他字典类属性），若是使用get这样的字典方法会比较好（字典的细节请参考第1章），因为这样一来即使你要查找的参数没有指定的话，程序的逻辑也不会出问题。

POST参数不属于URL的一部分，所以用户是看不到它们的，通常它们是Web页面上由HTML表单生成的。FORM标签的action参数指明了数据要提交给哪一个URL。用户提交表单的时候，URL就会随着由表单字段组成的POST字典一起调用。虽然从技术上说，它们的数据可以通过GET方法提交（很少见，因为没必要弄一个），不过绝大多数的Web表单都是用POST来操作的。

除了GET和POST，HttpRequest还有一个REQUEST字典，它会搜索前面两个字典试图返回一个被请求的键。如果一个给定的键值对可能被任何一个方法发送到视图里，你不知道改用哪一个的时候，这就很方便了。但是根据“尽量显式地表达意思而不要依赖于默认行为”的

Pythonic哲学，大多数有经验的Django程序员都会避免使用这个特性。

Cookies和会话（Sessions）

说完了GET和POST，请求对象里接下来最常用的就是request.COOKIES，它也是一个字典，代表了存储在请求里的HTTP cookies。cookies是网页里一种在用户浏览器里存放持久化信息的手段——大多数网站的认证系统都会用到它，有些商业站点也会用它来跟踪用户的浏览历史。

通常来说，大多数cookies都是用来支持一个叫做会话（session）的特性。这代表一个网页可以向浏览器要求一个能标识用户的值（这个信息可以在用户首次连接网站或是登录的时候设置），然后用这个信息来为那个用户提供定制行为的页面。由于cookies在客户端可以被轻易修改，所以存放关键数据是很不安全的，大多数网站都会选择把信息存放在一个服务器端的会话对象里（通常是站点的数据库）而只在cookies里保留一个唯一的会话ID。

会话通常是用来实现状态（state），因为HTTP协议本身是没有状态的（stateless）——每一轮请求/响应都是独立的，对之前或之后的请求完全没有概念。而有了会话以后，Web应用就可以绕过这个限制在服务器端保存数据（例如通知用户提交的表单是否成功保存了他的修改）并在稍后的响应里把它们发送给用户了。

Django里的会话也是HttpRequest对象的一个字典类属性，request.session（注意这里session是小写，而像其他属性那样是大写——这是因为会话并不是HTTP协议的一部分）。和之前的COOKIES属性一样，Python代码也能够读写session。当第一次被传递到代码里来时，它包含的是根据用户会话cookie从数据库里读出的会话数据。在写入的时候，它会把你做出的修改保存到数据库里，以便将来读取。

其他服务器变量

前面讲到的都是请求对象里最常用的属性，不过请求对象里还有很多其他变量，这些变量通常都是只读的，其中有些是HTTP标准的一部分，剩下的则是专为方便Django而准备的属性。下面列出了请求对象里所有可直接访问的属性（direct attribute）。

- path：URL里域名后的部分，例如，/blog/2007/11/04/。这个通常也是URLconf要处理的字符串。
- method：“GET”或“POST”两者之一，标明了这个请求用的是哪一个HTTP请求方法。
- encoding：标明了用来解码表单提交数据所需的编码字符集的字符串。
- FILES：这是一个字典类对象，包含了所有通过文件输入表单字段上传的文件，其中每一个值又都是另一个字典，里面包含了文件名、内容的类型以及文件的内容。
- META：这也是一个字典，它包含了所有没有被请求的其他部分处理的HTTP服务器/请求变量，例如CONTENT_LENGTH、HTTP_REFERER、REMOTE_ADDR和SERVER_NAME等。
- user：Django的认证用户，只有当你的站点激活Django的认证机制时才会出现。
- raw_post_data：请求里包含的POST原始数据。几乎都是推荐直接使用request.POST而不是去访问request.raw_post_data，不过若是有任何高级需求的话，也可以直接访问这里。

响应对象

到此，你已经读入传递给视图函数的信息了。现在我们来看看负责返回的响应。对我们来说，响应比请求要简单一点——其主要数据就是存放在content属性里的正文（body text）。一般就是一个巨大的HTML字符串，它对HttpResponse非常重要，你有好几种设置它的方法。

最常用的方法就是创建响应对象——HttpResponse接受一个字符串作为构造函数的参数，并将它保存在content里。

```
response = HttpResponse("<html>This is a tiny Web page!</html>")
```

这样你就有了一个完整的响应对象，可以返回给Web服务器并转发给用户浏览器了。不过，有时候需要一点一点的构建响应内容，HttpResponse对象实现了一个有点类似文件的行为来支持这一点，那就是write方法。

```
response = HttpResponse()
response.write("<html>")
response.write("This is a tiny Web page!")
response.write("</html>")
```

这就是说你可以把HttpResponse用在任何接受一个类似文件对象的代码里（比如csv模块里写CSV的工具），由此灵活地生成代码返回给最终用户的信息。

响应对象里的另一个功能就是设置HTTP头，这时你可以把HttpResponse当作是一个字典来用。

```
response = HttpResponse()
response["Content-Type"] = "text/csv"
response["Content-Length"] = 256
```

最后，Django为很多常见的响应类型提供了HttpRequest子类，例如HttpResponseForbidden（即HTTP 403）和HttpResponseServerError（即HTTP 500，或服务器内部错误）。

中间件

虽然Django应用程序的基本流程还是比较简单的（接受请求，找到适合的视图函数；返回响应），不过还可以在上面加诸更多层次来变得更灵活强大。其中之一就是中间件（middleware）——这些Python函数可以在上述过程里的多个地方执行来改变整个应用程序的输入（在请求到达视图之前对它进行修改）输出（修改视图创建的响应）。

Django里的中间件组件就是一个Python类，它实现了一个特定的接口，就是说它定义了一些名为process_request或是process_view这样的方法。（在下面几节里，我们会讨论其中最常用的一些。）在settings.py文件里的MIDDLEWARE_CLASSES元组里列出后，Django会内省这些类，并且在适当的时候调用其方法。类在设置文件里罗列的顺序也决定了它们执行的顺序。

Django自带了很多好用的内置中间件，有些可以直接拿来用，有些则需要特定的“contrib.”应用程序支持，比如认证框架。更多信息可以查阅Django官方文档。

请求中间件

请求中间件一般用在输入端，它定义为一个实现了process_request方法的类，像这样：

```
from some_exterior_auth_lib import get_user

class ExteriorAuthMiddleware(object):
    def process_request(self, request):
        token = request.COOKIES.get('auth_token')
        if token is None and not request.path.startswith('/login'):
            return HttpResponseRedirect('/login/')
        request.exterior_user = get_user(token)
```

注意给request.exterior_user赋值的那一行展示了请求中间件的一个常用用法：给请求对象添加额外属性。在那一行被调用后，process_request隐含返回None（如果没有显式的return语句，Python的函数一定返回None），然后Django会继续处理其他请求中间件，最后才轮到视图函数本身。

但是，如果对一个有效的认证符号（valid auth token，并且用户当前并没有要视图登录）测试失败的话，我们的中间件会把用户重定向到登录页上去。这也展示了中间件方法的另一种能力。它们可以返回一个立刻发送给请求客户端的HttpResponse（或子类）对象。在这里，因为我们的中间件是一个请求中间件，在那一行之后正常流程里的所有代码（包括原来要调用的视图）全部都会被忽略掉。

响应中间件

响应中间件是运行在视图函数返回的HttpResponse对象之上。这样的中间件必须实现process_response方法，这个方法接受一个request和一个response参数并返回一个HttpResponse或子类的对象。这些就是全部的限制了——你的中间件可以修改得到的响应或是创建一个全新的响应并返回。

响应中间件一个常见的用法就是在响应里插入额外的头信息（header），即可以是全面地（激活HTTP缓存等相关特性），也可以有选择地（把Content-Language设置为当前语言）。

下面是一个简单的例子，它把Web应用输出的所有文本里的“foo”都替换成“bar”：

```
class TextFilterMiddleware(object):
    def process_response(self, request, response):
        response.content = response.content.replace('foo', 'bar')
```

当然我们可选用比较实际的例子，如过滤所有的脏字（这对社区型的网站很有用）等，但为了不让小孩子学坏，还是免了吧！

5.3 视图与逻辑

视图（也叫控制器）是所有DjangoWeb应用程序的核心，因为它们提供了几乎所有实际的程序逻辑。在定义和使用模型的时候，我们是数据库管理员；在编写模板的时候，我们是界面设计师；而在编写视图的时候，我们才是真正的软件工程师。

虽然视图本身轻易就能占据你大部分源代码，但是Django框架里有关视图的代码量倒是相当有限。视图代表的是你的商业逻辑，所以对Web应用程序来说这部分应该需要最少的胶水代码，而大部分工作应交给程序员。同时，Django内置的通用视图（generic view）是Web开发框架里最能为你节省时间的工具之一，我们会和自定义视图一起详细介绍它们以及使用的方法。

就是Python函数

本质上来说，Django视图就是Python函数那么简单。对视图函数的唯一要求就是它们必须接受一个HttpRequest对象并返回一个HttpResponse对象，这两个在前面都介绍过了。还有就是前面提到的URLconf里的正则表达式模式，你可以在那里定义命名组（named group）。最后再加上一个可选的字典参数，就构成了视图函数所需的所有参数。像这样（和前面的例子稍有不同）：

```
urlpatterns = patterns('myproject.myapp.views',
    url(r'^archives/(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})/$', 'archive',
        {'show_private': True}),
)
```

算上HttpRequest对象，这个URL引用的archive视图函数的签名应该是这样的：

```
from django.http import HttpResponse


def archive(request, year, month, day, show_private):
    return HttpResponse()
```

只要它返回的是一个HttpResponse对象，方法里面做什么都行：我们在这里看到的只是一个API而已。对于任何API，你可以用已经写好的代码来实现它，也可以自己从头开始写。我们一个一个来看。

通用视图

Django里（甚至Web框架里）用的最多的功能大概就是使用预定义的代码来编写组成大多数Web应用的CRUD操作了。CRUD全称是Create（创建）、Read（读取，也叫Retrieve，获取）、Update（更新）和Delete（删除），它们是在一个数据库应用里用的最频繁的操作了。显示一列数据或是显示单个对象的细节页面？那就是Retrieve。显示一个编辑表单，提交后修改数据库？根据应用程序和表单的不同，这可以是Update或Create。Delete的意思就很清楚啦。

这些任务和变型都是为Django的通用视图准备的。刚刚已经展示过，它们全部都是Python函数而已，但是对于被定位的角色来说，它们又都是高度抽象和参数化的函数以达到最大的灵活性。因为它们负责处理逻辑，框架的用户只需要在他们的URLconf文件里引用它们，传递适合的参数，并确保视图要渲染和返回的模板存在就可以了。

举例来说，object_detail通用视图的作用是显示单个对象，并从URL正则表达式和参数字典里取得它的参数：

```
from django.views.generic.list_detail import object_detail
from django.conf.urls.defaults import *
```

```
from myproject.myapp.models import Person

urlpatterns = patterns('',
    url(r'^people/(?P<object_id>\d+)/$', object_detail, {
        'queryset': Person.objects.all()
    })
)
```

在上面的例子里，我们定义了一个正则表达式来匹配 /people/25/ 这样的URL，这里25代表的是我们要显示的Person记录在数据库里的ID。object_detail通用视图需要一个object_id参数和一个它可以通过ID过滤查找对象的QuerySet作为参数。在这里，我们通过URL提供参数，并通过字典参数提供 queryset。

将QuerySet整个传递给通用视图

传递Person.objects.all()看上去非常的不高效，因为就这样执行的话，QuerySet可能会返回一个巨大的列表包含了所有Person对象！不过还记得你在第4章里看到的吗——QuerySet可以（也通常）在真正执行数据库查询之前用filter或exclude排除一部分结果。因此，你可以放心object_detail通用视图会过滤掉特定的对象，把查询的结果控制在合理的范围内。

此外，要求一个完整的QuerySet而不是模型类（这是另一种指定我们感兴趣的对象类型的办法），Django允许我们自己进行过滤。例如，一个只显示员工细节的页面可以传递Person.objects.filter(is_employee=True)而不是Person.objects.all()给object_detail。

Django的核心团队希望留给你尽可能多的余地，哪怕最后的功能有时候乍看起来不太直观。

通用视图一般都提供了很多选项。有些是某个视图特有的，但是其他都是全局的，比如template_name参数允许用户重写视图模板的默认位置，而extra_context字典则允许用户向模板context传递额外的信息。（关于模板和context的细节请参阅第6章。）Django的官方文档上可以查到所有通用视图及其参数的细节，这里我们只讨论其中最常用的一些。注意为简洁起见，通用视图都是组成一个两层的模块层次。

- simple.direct_to_template：对含有动态内容（和简单页面flatpage，也就是静态HTML页面相对，详见第8章）但是没有特定Python逻辑的模板很有用，例如首页或是不分页/混合列表页面。
- list_detail.object_list和list_detail.object_detail：这两个提供了大多数Web应用里只读的部分，而且可能用的最多的通用视图，因为显示信息通常不需要什么复杂的逻辑。但是，如果你要执行一些逻辑来准备模板context的话，最好还是使用自定义视图吧。
- create_update.create_object和create_update.update_object：对简单对象的创建和更新很有用，你只需要在表单或是模型（分别参见第6章和第4章）里定义表单验证就行了，除此之外不需要任何其他商业逻辑。
- date_based.*：一组基于日期的通用视图，强调了Django原本是一个面向出版界框架的出

身。它们对任何基于日期的数据类型都非常有用。包括面向日期的索引和细节页面加上从年到日的子列表页面等。

通用视图也是一把双刃剑。其优点非常明显，它们能在简单视图到中等复杂的视图里，为你节约大量的时间和绝大部分的工作。通过把它们包裹在自定义视图里，还能进一步拓展它们的能力，下面我们就要说到这个话题。但是再强大它也有局限的时候，有时候你还就非得自己重新写一个自定义的视图出来不可，哪怕某个通用视图已经能满足就90%的要求了。知道什么时候放弃而采用自定义视图是一项很有价值的技能，和软件开发的很多方面一样，只能通过经验慢慢积累。

半通用视图

有时候直接从URLconf文件里调用通用视图还不太够用。一般这种情况下，只能重新写一个自定义的视图，但同时，根据需要通用视图还是可以用来帮助你干一些底层的体力活的。

在我们的经验里，这种“半通用视图”最常见的用法就是用来绕过URLconf自身固有的限制。你不能在正则表达式解析之前对捕捉到的URL参数进行任何操作。这个限制是由URLconf的设置所决定的，不过要绕过它也不难。请看下面这段URLconf文件和视图文件的代码：

```
## urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('myproject.myapp.views',
    url(r'^people/by_lastname/(?P<last_name>\w+)/$', 'last_name_search'),
)

## views.py

from django.views.generic.list_detail import object_list
from myproject.myapp.models import Person
def last_name_search(request, last_name):
    return object_list(request,
        queryset=Person.objects.filter(last__istartswith=last_name)
    )
```

可以看到，虽然函数接受了一个在URL正则表达式的命名组里定义的last_name参数，但是我们还是把99%的实际工作交给了通用视图去处理。之所以可以这么做是因为通用视图就是普通的Python函数，一样可以导入和调用。人们很容易陷入一种怪圈，认为框架自身是一种语言，其实就和我们前面多次强调的那样，所有一切都是Python，并且刚才这种技巧更显出为什么这种设计思想是出色的。

自定义视图

最后，我们前面说过，有时候就是没办法使用通用视图，还记得本节开始的地方么，视图函数本身只是一块空白画布，它只是遵循了一个简单的API，你可以往里面填充任何东西。这

里我们将和你分享一点我们自己的经验，并指点一些框架提供的方便快捷的函数给你。不过总的来说，在这块领域里，只有你自己的技术和经验才能决定下一步怎么走。

框架提供的捷径

就像刚才说的，一旦开始编写自定义视图，Django基本上就不再对你做任何限制了。但是，它还是提供了一些捷径给你，其中绝大多数都属于django.shortcuts模块。

- render_to_response：这个函数替代了两步到三步的过程，从创建一个Context对象，然后用Template渲染，最后返回包含结果的HttpResponse。它接受一个模板的名字，一个可选的context对象（也可以是字典）以及MIME类型，并返回一个HttpResponse对象。模板渲染会在第6章里介绍。
- Http404：这个Exception子类会返回一个HTTP 404错误码并且渲染一个顶层的404.html模板（除非你在settings.py里重写它）。使用起来和raise其他异常没什么两样，所以当你遇到一个404的时候，它就是一个标准的错误，就和你试图把一个字符串加到一个整数上时发生的错误是一样的。这个类定义在django.http模块之下。
- get_object_or_404和get_list_or_404：这两个函数就是把两个步骤合二为一：获得一个对象或者列表，如果查找失败则抛出Http404。它们接受一个klass参数（它很灵活，能够接受一个模型类，一个Manager，或者一个QuerySet）和一些数据库查询参数，例如那些要传递给Manager和QuerySet的参数，并试图返回所需的对象或者列表。

这里是两个使用这些快捷方式的例子：第一个直接使用Http404，而第二个则展示了如何用get_object_or_404来一步完成同样的目标——这两个函数在实际中的行为完全相同。先不要管这里的模板路径，在第6章里会详细解释它们的。

这里是手动抛出404异常的方法：

```
from django.shortcuts import render_to_response
from django.http import Http404
from myproject.myapp.models import Person

def person_detail(request, id):
    try:
        person = Person.objects.get(pk=id)
    except Person.DoesNotExist:
        raise Http404

    return render_to_response("person/detail.html", {"person": person})
```

这个例子则是使用get_object_or_404，通常你都会用这个方法来替代前面那个：

```
from django.shortcuts import render_to_response, get_object_or_404
from myproject.myapp.models import Person

def person_detail(request, id):
    person = get_object_or_404(Person, pk=id)

    return render_to_response("person/detail.html", {"person": person})
```

其他观察

很多Django程序员发现他们在定义自己的视图函数时总是会用到“args/kwargs”方式。在第1章里就说过，Python函数定义*args和*kwargs后就能够接受任何位置参数和关键字参数。虽然这是一把双刃剑（出色的文档通常源自明确的函数签名，不过这里就没有了），但是这确实是一个很好用的技巧，既灵活又快捷。你不用再回到URLconf文件里去记住你是怎么给捕捉正则表达式参数或是关键字参数命名的，你只需要这样定义函数：

```
def myview(*args, **kwargs):
    # Here we can refer to e.g. args[0] or kwargs['object_id']
```

然后什么都不用管，直接在需要的时候引用kwargs["identifier"]就行了。不知不觉地你就会不由自主地使用它，当你要把函数的参数传递给一个代理函数的时候（例如在前面的“半通用”视图里提到的）也会容易得多。

5.4 总结

到此我们探索Django基础核心组件的旅程就已经完成一半了。除了在第4章里描述的模型(model)，这一章向你展示了Django是如何实现URL转发以及HTTP请求-响应的“对话”，包括中间件的使用。你还看到了如何把简单的Django视图函数组合在一起，另外还感受了通用视图以及使用。

下一章也是本书里这部分的最后一章，第6章描述了框架里第三个主要的部分，即通过模板渲染网页以及用表单和表单验证管理用户的输入。之后，你就会进入到第三部分，看看这些概念是如何在那4个示例应用程序里使用的。

第6章 模板和表单处理

学习完Django的数据模型和逻辑处理后，我们还剩下最后一个部分：如何显示信息和管理用户输入。我们先介绍一下Django的模板语言和渲染系统，然后在第二部分介绍表单和表单处理。

6.1 模板

在之前的章节里已经我们已经接触过，模板是一种独立的文本文件，同时包含了静态内容（比如HTML）和动态标记的逻辑、循环和数据的显示等。使用哪个模板以及渲染什么数据是由视图函数本身（通过显式地渲染或者使用render_to_response）或者视图的参数（比如通用视图里的template_name参数）决定的。

Django的模板语言是设计给前端工程师使用的，他们不一定是程序员。正因为如此，再加上将逻辑和表现分开的愿望，模板语言绝对不是嵌入式的Python。不过，Django程序员还是可以通过可扩展的标签（tags）和过滤器（filter）系统来拓展模板语言的逻辑结构（后面会说到）。

最后，请注意虽然模板系统通常被用来生成HTML（它就是Web嘛），不过它并没有被绑定给HTML，而是可以用来生成log文件、E-mail正文、CSV文件等任何文本格式。别忘了这一点，它能帮助你充分挖掘Django模板的潜力。

理解Contexts

模板作为一种动态文档，只有在显示的动态信息的时候才能证明自己的存在。Django里把传给一个渲染模板的信息成为context——一个模板的context基本上就是一个包含键值对的字典，在渲染的时候表示了一个类似字典的Context对象。

第2章和第5章里我们都看到过了，每渲染一个模板都需要有一个context。有时候context会为你事先准备好（比如在通用视图里），你只要把extra_context参数附加上去就行了。而其他时候（比如自定义视图），你就得在调用模板render方法的时候，自己准备context，或者更常见的是作为参数传给帮助函数render_to_response。技术上来说，你可以用空context来渲染一个模板，不过如果是这样的话，你应该考虑去用flagpages contrib应用程序——毕竟一个没有context的模板就不是那么动态了。

另一个给模板context提供数据的方法就是通过context处理器（processor），这是框架里一个类似中间件的部分，你可以在那里定义各种函数，在模板渲染之前来把键值对附加到所有context上去。这就是认证框架这样的特性为什么能保证特定的数据在全站范围里都能访问到的原因。这里是一个context处理器的例子：

```
def breadcrumb_processor(request):
    return {
        'breadcrumbs': request.path.split('/')
    }
```

看上去好像不太有用（网站指引很少有这么简单的），不过它展示了context处理器的简单之处。你可以把context处理器函数放在任何地方，不过通常标准化总是有好处的，比如context_processors.py文件可以放在项目的根目录里，或者是应用程序的目录里。

Context处理器（和中间件一样）需要在settings.py里用Python模块语法引用才能激活，它的位置是一个名叫TEMPLATE_CONTEXT_PROCESSORS的元组。另一个和中间件相似的地方就是它们在元组里指定的顺序是有讲究的，context处理器会按照在设置变量里的顺序逐个应用。

模板语言语法

Django模板语言的语法可以和其他一些非XML的模板语言（比如Smarty或是Cheetah）比较起来的区别在于，它没有要保持XHTML兼容的意思，而是用特殊的字符来把模板变量以及逻辑命令和静态内容（通常是HTML）分开。和Django的其他部分一样，模板语言和框架的其他部分都是松耦合的，所以只要你愿意，完全可以换用另一种模板库。

大多数模板语言都有单独的命令（singular command，比如打印一个context变量的值）和模块级命令（block-level command），通常是逻辑命令“if”或者“for”等。Django的模板语言有两个约定用法，都和大括号有关。变量输出用的是双大括号（{{ variable }}），而其他则是用标签（{% command %}）。这里是一个渲染类似Python字典（{ "title_text": "My Webpage", "object_list": ["One", "Two", "Three"] }）的context的例子。

```
<html>
    <head>
        <title>{{ title_text }}</title>
    </head>
    <body>
        <ul>
        {% for item in object_list %}
            <li>{{ item }}</li>
        {% endfor %}
        </ul>
    </body>
</html>
```

注意当你在模板里输出context变量的时候，它会隐式地调用unicode方法，所以对象以及其他非字符串变量都会被尽量转换成（Unicode）字符串。小心——如果你试图打印没有定义__unicode__方法的对象，在模板里是看不到它们的。这是因为Python默认表示一个对象的形式正好和HTML的标签格式相同，即文本都包含在 < 和 > 尖括号里。

```
>>> print object()
<object object at 0x40448>
```

即便是有经验的Django程序员有时候也会踩到这个常见的陷阱里去，所以如果你发现要显示的东西没显示出来，第一个要确认的就是那是什么对象以及那个对象的字符串表示是什么！

你可以看到，虽然Django模板的语法不是语义正确的HTML，但是大括号的语法很容易就能让人分辨出哪里是输出和命令的部分，哪里是静态的内容。此外，Django的开发团队打算的是用模板语言生成文档类型而不仅仅是HTML，所以他们觉得设计一个针对XML输出的模板系统是行不通的。

模板过滤器

虽然变量输出是构建动态模板的基础，但是它实在是太不灵活了。模板框架可以通过一种叫过滤器（filter）的机制来转换context变量，它和UNIX里的管道有点相像——要是你不知道什么是管道的话，请参阅附录A。过滤器连语法都是直接照搬UNIX管道，它用的也是管道符(|)。因为它们总是接受一个文本输入并返回一个文本输出，所以可以把它们串在一起使用。在第11.5节里你会看到这种手法，过滤器也是Python函数。

Django提供了各种各样有用的过滤器来封装Web开发中常见的文本处理工作，例如转义斜杠符号、大写首字、格式化日期、获取列表或元组的长度，以及连接字符串等。这里一个如何使用过滤器来把一个字符串列表转成小写的例子。

```
<ul>
{% for string in string_list %}
    <li>{{ string|lower }}</li>
{% endfor %}
</ul>
```

虽然大多数过滤器值接受一个字符串输入，不过有些也能进一步地参数化其行为，比如yesno过滤器可以接受任何值（一般是布尔值）并且打印出可读的字符串。

```
<table>
    <tr>
        <th>Name</th>
        <th>Available?</th>
    </tr>
    {% for person in person_list %}
    <tr>
        <td>{{ person.name }}</td>
        <td>{{ person.is_available|yesno:"Yes,No" }}</td>
    </tr>
    {% endfor %}
</table>
```

标签

在上面的例子你可能已经注意到了，虽然变量输出和过滤器都很有用，但是真正强大的确实是标签（tag）——到目前为止我们已经见过它们用来循环字符串或对象的列表，但是它们还能够执行逻辑操作（{% if %}，{% ifequal %}），模板包含/继承（{% block %}、{% include

%}和{% extends %}，参见下一节），等其他一系列任务。

标签从技术上来讲是不定型的，可以在标签名后面跟任何形式的输入（细节请参考第11.5节），但是内置标签和绝大多数用户创建的标签都尽量遵循一定的约定，通常是一列由空格分开的参数。很多标签参数可以是context变量，而且事实上大多数时候，还可以用过滤器。比如，下面的例子是如何在迭代列表的内容前检查它的长度。

```
{% ifequal object_list|length 10 %}
    <ul>
    {% for item in object_list %}
        <li>{{ item }}</li>
    {% endfor %}
    </ul>
{% endifequal %}
```

当然我们还可以用length_is过滤器，它接受一个列表和参数并返回一个布尔值。

```
{% if object_list|length_is:10 %}
    <ul>
    {% for item in object_list %}
        <li>{{ item }}</li>
    {% endfor %}
    </ul>
{% endif %}
```

希望这个例子展示出了Django内建的过滤器和标签库的灵活之处。事先多知道一点有什么可用（Django官方文档里提供了一份出色的列表）就能避免重新发明轮子啦。

最后关于标签的一点：模块级命令（block-level）如{% if %}和{% for %}可以修改它们的局部context，这个特性通常都很好用。例如，{% for %}提供了一个局部context变量{{ forloop }}，它提供了各种属性允许你根据使用的属性和所处的循环迭代来进行不同的操作。这种属性相关的操作有，在循环开始和结束的时候（{{ forloop.first }} 或 {{ forloop.last }}分别返回布尔值告知这是不是循环里的第一个或是最后一个迭代）执行什么操作，或者是显示循环计数（{{ forloop.counter }} 和 {{ forloop.counter0 }}，分别是从1或0开始）等。更多信息和例子请参考Django官方文档。

Blocks和Extends

模板里有一组很有用的标签，它们可以跃出当前的模板和其他模板文件交互，通过继承（inheritance）和包含（inclusion）来完成代码组合和重用。我们先介绍继承，因为它通常更有助于逻辑化的组织模板。而包含，虽然也很有用，却也很容易导致所谓的“include soup”，让调试和开发变得很困难。

模板继承是通过{% extends %}和{% block %}两个模板标签实现的。{% extends %}必须在模板的顶部调用，并告知渲染引擎这个模板是从一个更高级的模板继承而来。例如，你可以定义一个顶层的或全站可见的模板标出页眉/页脚和全局引导，然后在每个子小节的中层模板里，扩展这个顶层模板（例如添加第二层的菜单指引），最后，每个单独的站点里的底层模板再扩

展中层的模板来提供页面的实际内容。

{% block %}是一个模块级的标签，可以用来定义模板里预备让那些要扩展它的模板去填充的小节。虽然模板的儿子会通常都会使用这些块（block），但其实这不是必需的。它们可以被忽略掉（这样父模板里有什么它就会显示什么）或是进一步代理给更低一层的模板。下面这个简单的例子使用了之前提到的三层网站布局，它的URL有/、/section1/、/section2/、/section1/page1/和/section1/page2/。

我们暂时先忽略根站点的首页，而专注于最底层的“叶子”页面。这里，base.html提供了顶层的封装结构，section的模板提供了页面的标题（指明了用户所处的是站点的哪一部分），而page的模板则提供了simple content。

base.html:

```
<html>
    <head>
        <title>{% block title %}My Web site{% endblock %}</title>
    </head>
    <body>
        <div id="header">
            <a href="/section1/">Section 1</a>
            <a href="/section2/">Section 2</a>
        </div>
        <div id="content">
            {% block content %}{% endblock %}
        </div>
        <div id="footer">
            <a href="/about/">About The Site</a>
        </div>
    </body>
</html>
```

section1.html:

```
{% extends "base.html" %}

{% block title %}Section 1{% endblock %}
```

section2.html:

```
{% extends "base.html" %}

{% block title %}Section 2{% endblock %}
```

page1.html:

```
{% extends "section1.html" %}

{% block content %}This is Page 1.{% endblock %}
```

page2.html:

```
{% extends "section1.html" %}

{% block content %}<p>This is Page 2.</p>{% endblock %}
```

前面例子里的模板都设置好后，用户在访问/section1/page2/的时候就能在浏览器里看到下面的内容：

```
<html>
    <head>
        <title>Section 2</title>
    </head>
    <body>
        <div id="header">
            <a href="/section1/">Section 1</a>
            <a href="/section2/">Section 2</a>
        </div>
        <div id="content">
            <p>This is Page 2.</p>
        </div>
        <div id="footer">
            <a href="/about/">About The Site</a>
        </div>
    </body>
</html>
```

模板继承的优点就在于很容易在模板层次里穿梭来找到什么模板生成了任何给定页面里的哪部分HTML。此外，和基于包含的那种方式即在每个子页面里都要包含页眉、页脚、边栏等代码的方式比起来，继承能够节省很多代码量。

包含其他模板

尽管模板继承有这么多优点，不过模板包含依然有其立足之地。有时候你需要重用的一块HTML或者其他文本没法很好的符合继承的需要，比如一个常用的分页元素。Django支持{% include %}包含（用法完全可以猜到），它接受要包含的模板文件的名字，然后把自己替换成这个文件的内容。包含文件自己就可以是一个完整实现的Django模板，它们的内容是遵循包含模板的context来解析的。

除了{% include %},Django还提供了{% ssi %}标签（这里的ssi代表的 Apache家族的SSI，即服务器端包含，Server Side Includes)。{% include %}和{% extends %}引用的是在settings.py里指定的template目录中定义的局部模板文件。而{% ssi %}使用的则是文件系统绝对路径。不过为了安全期间，{% ssi %}被限制在一组特定的目录里，它可以通过settings.py文件里的ALLOWED_INCLUDE_ROOTS变量指定。

最后还要注意{% extends %}和{% include %}既可以接受context变量也能接受字符串，这就等于模板可以动态地决定它们要包含的模板或是从什么模板继承了。

6.2 表单

虽然模板很适合把信息显示出来，但是把它输入到数据库就是完全另外一码事了，这需要创建并验证HTML表单以及保存提交的信息。Django提供了forms库来把框架里三个主要的组件联系在一起：模型里定义的数据库字段，模板里显示的HTML表单标签，还有检验用户输入和显示错误信息的能力。

在编写本书的时候，Django的表单机制还处在一个不断变化的过程中，这里我们讨论的库现在叫做newforms，它是一个模块化的方法，比它的先驱oldforms要进步的多了（虽然现在import django.forms返回的还是这个旧的库）。我们在这里讨论的是newforms，并且引用（和导入）它的时候都是用现在时态forms，希望在你读到这本书的时候，它已经完成转变，成为Django里表单处理的前线主力了。

定义表单

表单处理的核心是一个和Model相似的类：Form。和模型一样，表单其实也是变量对象的集合，除了它们代表的是一个特定的Web输入表单而不是一张数据表。很多时候我们感兴趣的是和给定模型100%匹配的表单，不过有一个分开的Form类也在一定程度上增加了灵活性。

有了这个单独的实体，你可以隐藏或是省略特定的变量，或是把多个模型类里的变量放在一起。当然，有时候你还会希望可以处理和数据库存储完全没关系的表单，这些全都可以通过Form类完成。我们先来看一个简单的例子。

```
from django import newforms as forms

class PersonForm(forms.Form):
    first = forms.CharField()
    last = forms.CharField()
    middle = forms.CharField()
```

虽然这个看起来有点疑似之前的模型类，不过它完全是一个独立的表单，只不过它们正好有着相同的变量而已（除了这些都是forms.Field实例而非models.Field实例）。表单变量接受的参数和模型里也是非常的类似。

```
class PersonForm(forms.Form):
    first = forms.CharField(max_length=100, required=True)
    last = forms.CharField(max_length=100, required=True)
    middle = forms.CharField(max_length=100)
```

上面的例子定义了一个由三个文本变量组成的表单，在验证的时候（见下面），如果first和last变量没有被填入的话它就会报错。此外它还保证这些变量都不会超过100个字符长。因此数据库字段类型和表单变量类型有很多重叠的地方——如果你想知道在本章的示例之外还有什么的话，请查阅官方文档里表单Field类的详细列表。

基于模型的表单

为了保持DRY原则，Django允许你使用一种特殊的Form变型ModelForm来为任何模型类或实例取得一个Form子类。一个ModelForm和一个普通的Form基本上一模一样，但是它包含了一

个Meta嵌套类（和模型类里的Meta类似），Meta类里有一个必需的属性model，它的值就是所需的Model类。下面的例子和前面定义的普通Form在功能上是一致的：

```
from django import newforms as forms
from myproject.myapp.models import Person

class PersonForm(forms.ModelForm):
    class Meta:
        model = Person
```

一般你需要为你创建的每一个模型类都至少定义一个ModelForm，哪怕它简单到这个例子这样。这个方法突出了数据定义和数据输入和验证的分离，并且提供了很大的灵活性。

使用ModelForm后它会把你Model类里的变量“复制”到Form变量里来。通常这个过程都是很直观的（一个模型CharField变成了一个表单TextField或是ChoiceField，如果它定义了choices的话），除了Django官方上专门列出的一些警告。此外要记住的最重要的事就是变量默认都是required=True的，除非模型把它们设置为blank=True，这时它们才变成可选的变量（required=False）。

保存ModelForm

用这种方式生成的表单和手动生成的表单有一个重要的区别，就是它们有一个save方法，加入验证成功的话，可以把它们的信息保存为数据库里的一条记录，然后返回Model对象结果。一旦你读到了如何把信息填入到表单并且如何验证它的时候，这个方法的含义就会变得更明显起来，但是现在只要记住save方法能进化POST字典到数据库创建（或更新）的步骤。继续前面的例子（这个过程的细节会在本章稍后讲到）：

```
from myproject.myapp.forms import PersonForm

form = PersonForm({'first': 'John', 'middle': 'Quincy', 'last': 'Doe'})

new_person = form.save()

# Will result in the __unicode__() output for the new Person
print new_person
```

你经常会需要在数据从表单提交后到它存储到数据库之前修改它们。有时候你可以在POST字典被交给表单之前就修改它；但是有时候在表单完成验证后再修改可能会更简单一点（不过还是要在它到达模型层之前）。后面这种方式一般来说更好，因为这个时候原始的POST数据已经被转换成Python值了。

为了提供这类灵活性，save方法接受一个可选的commit参数（默认是True），它可以控制是不是真的要更新数据库。设置为False后它还是会返回模型对象给你，不过这是你就要自己负责调用对象的save方法了。这个例子只访问了一次数据库而不是两次，这只有当commit=False的时候才会发生。

```
form = PersonForm({'first': 'John', 'middle': 'Quincy', 'last': 'Doe'})
```

```
# We get a Person object, but the database is untouched.
new_person = form.save(commit=False)

# Update an attribute on our un-saved Person.
new_person.middle = 'Danger'

# Now we can update the database for real.
new_person.save()
```

另一个和commit=False有关的场景是当你使用内联编辑的相关对象的时候。这时，一个ModelForm会同时验证并保存主对象和相关对象的数据。因为关系数据库要求目标记录必须在引用之前就存在，所以不可能在保存相关对象的时候却延迟保存你的主要对象。

所以，当一个ModelForm包含了相关对象的信息，并且你又使用了commit=False时，Django会给表单（不是Model对象结果！）添加一个额外的方法save_m2m，它让你能正确地安排事件的顺序。在这个例子里，假设Person模型有一个自引用的多对多关系。

```
# This input to PersonForm would contain "sub-forms" for additional Person
# objects related to the primary one via the ManyToManyField.
form = PersonForm(input_including_related_objects)

# Those related objects can't be saved at this point, so they are
# deferred till later.
new_person = form.save(commit=False)

# Update an attribute on our un-saved Person.

new_person.middle = 'Danger'

# After we save to the DB, our Person exists and can be referenced by
# the related objects.
new_person.save()

# So now we save them as well. Don't forget to call this, or your related objects
# will mysteriously disappear!
form.save_m2m()
```

你可以看到，考虑立刻保存还是延迟保存的需求给save方法的使用增加不少复杂度。好在这种复杂度不是必需的，绝大多数情况下你都可以直接save而无需过多担心。

区别于Model

有时候你会希望自己定义表单，而不是作为一个模型的复制品。隐藏特定的变量是一个很常见的需求，稍微少见一点的是排除或是包含所有变量。通常你不需要重新写一个Form子类出来，因为有好几种方式用ModelForm来完成这个任务。

ModelForm的Meta嵌套类允许你定义两个可选的属性，fields和exclude，这两个是要包含或是排除的变量名的列表或是元组（当然了，这两个你每次只能用其中一个！）。例如，下面的例子里你定义了一个忽略了中间名的Person表单：

```
from django import newforms as forms
```

```
from myproject.myapp.models import Person

class PersonForm(forms.ModelForm):
    class Meta:
        model = Person
        exclude = ('middle',)
```

如果一个Person只有first、middle和last三个变量，那下面用fields实现的例子和上面这个功能一模一样：

```
class PersonForm(forms.ModelForm):
    class Meta:
        model = Person
        fields = ('first', 'last')
```

这里要注意的非常重要的一点是，当调用这样的表单的save方法时，它只会试图保存它知道的变量。如果你忽略了一些变量而模型里它们又必须出现的话这就会产生问题了。所以对这样的变量一定要再三确认它们要么被标志为可选的（null=True），要么在default参数里定义了默认值。

除了决定要显示模型里的哪些变量，你还可以重写forms层里的Field子类，它负责验证和显示特定的变量。做法很简单，直接显式地定义它们即可，就像在本章开头的地方看到的那样，它会自动重写从模型里取出来的定义。这对传给表单层的Field（比如max_length或required）或者是修改类本身（可能是要把TextField显示为CharField，又或者通过把一个CharField变成ChoiceField来添加一些选择等）都很有用。例如，下面的例子就把名字变量的长度收紧了一点：

```
class PersonForm(forms.ModelForm):
    first = forms.CharField(max_length=10)

    class Meta:
        model = Person
```

这里还值得一提的是关系表单变量（relationship form fields），ModelChoiceField和ModelMultipleChoiceField，它们分别对应ForeignKey和ManyToManyField。虽然你可以在模型层变量里指定limit_choices_to参数，其实你还可以在表单层变量里自定一个queryset参数（很自然的，它接受的是一个特定的QuerySet对象）。这样，你就可以在模型层重写这个限制（或缺陷）并且自定义ModelForm。就像下面的例子里，我们假设Person模型有一个指向其他Person对象的没有限制的父ForeignKey：

```
# A normal, non-limited form (since the Model places no limits on 'parent')
class PersonForm(forms.ModelForm):
    class Meta:
        model = Person

# A form for people in the Smith family (whose parents are Smiths)
class SmithChildForm(forms.ModelForm):
    parent = forms.ModelChoiceField(queryset=Person.objects.filter(last='Smith'))
```

```
    class Meta:
        model = Person
```

继承表单

很多时候，不管是普通的Form还是ModelForm，你都需要利用Python面向对象的特点来避免重复。Form的子类还可以继续被继承，其结果会包含父类里的全部变量。例如：

```
from django import newforms as forms

class PersonForm(forms.Form):
    first = forms.CharField(max_length=100, required=True)
    last = forms.CharField(max_length=100, required=True)
    middle = forms.CharField(max_length=100)
class AgedPersonForm(PersonForm):
    # first, last, middle all inherited
    age = forms.IntegerField()

class EmployeeForm(PersonForm):
    # first, last, middle all inherited
    department = forms.CharField()

class SystemUserForm(EmployeeForm):
    # first, last, middle and department all inherited
    username = forms.CharField(maxlength=8, required=True)
```

还可以“混合”操作，即多重继承。

```
class BookForm(forms.Form):
    title = forms.CharField(max_length=100, required=True)
    author = forms.CharField(max_length=100, required=True)

class InventoryForm(forms.Form):
    location = forms.CharField()
    quantity = forms.IntegerField()

class BookstoreBookForm(BookForm, InventoryForm):
    # Has title, author, location and quantity
    pass
```

在用这种方法继承ModelForm的时候，记住你还可以修改Meta的属性，通常是更新或是往fields或exclude里添加新的值来进一步限制可用的变量。

填写表单

在Django的表单库里，任何给定的表单实例要么是绑定的（bound），即和数据有关系的，要么是没绑定的（unbound），即没有数据的。没绑定的空表单主要是用来生成空的HTML表单来让用户填写，因为你没办法验证它们（除非一张空白的表单是所需要的输入，这种情况几乎没有），而且基本上也不会想要把它们的内容保存到数据库里去。绑定的表单则是大多数操作的所在。

数据绑定到表单是发生在实例化的时候，并且一旦完成实例化，表单就不能改变了。这听起来有点不灵活，但是这和允许表单改变比起来，能让使用和应用表单的过程显得更加明确和正交。它还消除了关于验证数据被改变的表单的状态时任何隐晦的地方（这在oldforms库里是有可能出现的情况）。

我们来根据前面和Person关联的ModelForm子类生成一个绑定的表单，把它嵌入到一个可以变成表单处理视图函数的开头。Request.POST.copy()在这里不是必需的，但是我们推荐使用它。在万一需要的时候，你可以在保持请求里原有内容不变的同时修改自己的字典拷贝。

```
from myproject.myapp.forms import PersonForm

def process_form(request):
    post = request.POST.copy() # e.g. {'last': 'Doe', 'first': 'John'}
    form = PersonForm(post)
```

需要注意的是你可以随便往表单的数据字典里添加额外的键值对，表单会自动无视那些和它们定义的变量没关系的输入。这就是说你可以拿一个更大的表单里的POST字典来用它填充一个代表那些变量子集的Form对象。

你还可以创建一个虽然未绑定，却在模板打印的时候载入显示了初始值的表单。这个命名构造函数参数（named constructor argument）initial是一个字典，和用于绑定的位置参数（positional argument）是一样的。每个表单变量都有这样一个类似的参数，允许它们指定自己的默认值，不过要是有冲突的话，表单级字典会覆盖掉它们。

这里是一个创建表单的例子，它修改之前自定义的PersonForm，这里把姓氏变量预填为“Smith”（通过表单定义）并在运行时把名字变量设为“John”（创建表单实例时）。当然，用户可以在填写表单的时候修改它们任何一个的值。

```
from django import newforms as forms
from django.shortcuts import render_to_response

class PersonForm(forms.Form):
    first = forms.CharField(max_length=100, required=True)
    last = forms.CharField(max_length=100, required=True, initial='Smith')
    middle = forms.CharField(max_length=100)

def process_form(request):
    if not request.POST: # Display the form, nothing was submitted.
        form = PersonForm(initial={'first': 'John'})
        return render_to_response('myapp/form.html', {'form': form})
```

注意

如果用来实例化PersonForm的initial参数是{'first': 'John', 'last': 'Doe'}，那么实例层上('last'键）的值“Doe”就会覆盖掉类定义层上的表单定义里的“Smith”。

使用实例层initial参数的好处在于它的值可以在表单创建的时候才被构造出来。例如，这

让你得以引用在表单或是模型定义的时候还不知道的信息，特别是请求对象里的信息。

我们来看看如何在这样的一个视图函数里利用这个特性，这个视图函数负责添加新的Person记录，而这个Person记录又要和另一个这样的记录有关联。假设在Person模型里有一个新的自引用的父ForeignKey，我们来为Person定义一个简单的ModelForm。

```
from django.shortcuts import get_object_or_404, render_to_response
from myproject.myapp.models import Person, PersonForm

# View's URL: /person/<id>/children/add/
def add_relative(request, **kwargs):
    # Display the form if nothing was POSTed
    if not request.POST:
        relative = get_object_or_404(Person, pk=kwargs['id'])
        form = PersonForm(initial={'last': relative.last})
        return render_to_response('person/form.html', {'form': form})
```

为简单起见，我们省略了一个视图函数真正关心的东西，表单提交的处理。注意这里我们是如何根据URL里的值来获取relative对象，然后把relative的姓氏作为表单里last的初始值传递进来的。换句话说，我们把数据都准备好，这样孩子会自动填写好父亲的姓氏——如果你的用户要输入很多数据的话，这个功能或许会很有用。

验证和清理

虽然表单通常是没有状态的，不过要是绑定表单的话，它们确实需要一些机制来验证绑定给它们的数据（验证和清理不能应用到非绑定表单上）。要让表单运行它的验证程序，你可以显式地调用is_valid方法，或是调用它的显示方式（见下），它们也会隐式地进行验证。

我们来重新安排一些刚才的add_relative表单处理视图，这样它就可以处理表单输入以及显示空表单。这需要把逻辑改得更加灵活一点，一个Django惯用的手法就是先检查POST字典是否存在并且处理验证（或者生成一个空的表单），然后“继续”显示表单。所以表单显示的要么是非POST的请求或者是验证失败的POST请求。

```
# View's URL: /person/<id>/children/add/
def add_relative(request, **kwargs):
    # Get the parent relative

    # Validate if the form was POSTed
    if request.POST:
        form = PersonForm(request.POST)
        if form.is_valid:
            new_person = form.save()
            return HttpResponseRedirect(new_person.get_absolute_url())
# Otherwise, prep an empty form with the relative pre-filled
else:
    relative = get_object_or_404(Person, pk=kwargs['id'])
    form = PersonForm(initial={'last': relative.last})
# Display the form for non POST requests or failed validations.
```

```
    #'Our template will display errors if they exist.
    return render_to_response('person/form.html', {'form': form})
```

当执行验证后，表单对象就会得到这两个新属性之一：一个包含错误信息的字典errors，或是一个包含了原本绑定到表单上的值的“干净”版的字典cleaned_data。这两个属性永远都不会同时出现，因为cleaned_data只有在表单验证通过的时候才会生成，而只有验证失败的时候errors才会被应用。

Errors字典的格式非常简单，键是变量的名字，值是字符串的列表（其中每个字符串是一条关于为什么表单验证失败的消息）。通常errors里只包含了有错误要显示的变量键值对。在本章的稍后，我们会讨论一些Form对象提供的帮助函数来方便地显示这些错误详细。

这些“干净”数据背后的意义在于输入的数据需要规范化——即从一种或多种可能的输入格式转换为一个统一的输出格式，以方便验证和数据库存储。例如，表单里绑定数据的request.POST字典里通常包含的是字符串，所以任何数字变量的清理过程就需要把那些字符串变成int或long，日期相关的变量则要把“2007-10-29”这样的字符串解析成为datatime对象等。

虽然自动验证和保存方法都需要数据正规化才能正常工作，它还意味着任何和表单内容打交道的Python代码都能访问到正确的数据类型。当然如果你发现有需要去检验原始数据的话，它们都保存在表单的data属性里面。

显示表单

表单对象提供了一些很有用的方法来帮助你把它们用不同的预定义HTML格式显示出来。这些方法打印出表单的整个内部结构，不带<form>标签、提交按钮，或<label>标签。如果你需要更细致地控制输出，还可以单独地显示每一个表单变量。最后，虽然这些显示方法大多是在用在模板里面，但这一点并不是强制的——要是你有需要，完全可以在Python代码里调用它们任何一个甚至全部。

每个Django表单变量都知道在HTML标签里要怎么显示自己，这种行为是通过widgets来改变的，在本章的最后会介绍到它。另外，这些标签的name和id属性，以及它们对应的<lable>标签里的for属性，都是取自一开始你定义的Form类里的变量属性名。<lable>标签里的文本默认是这样生成的：将变量名首字大写，把下画线换成空格，要是变量名不是以标点符号开始的话，就再加上一个结尾的:字符。

这里是一个例子来帮助你理解这些不同的选项以及它们是如何影响HTML输出的。首先，我们重新复习一下这个手动创建的PersonForm：

```
from django import newforms as forms


class PersonForm(forms.Form):
    first = forms.CharField(max_length=100, required=True)
    last = forms.CharField(max_length=100, required=True, initial='Smith')
    middle = forms.CharField(max_length=100)
```

然后是first变量显示在表格里的一部分时是怎么转换成HTML的：

```
<tr><th><label for="id_first">First:</label></th><td><input id="id_first"
type="text" name="first" maxlength="100" /></td></tr>
```

你还可以通过表单的auto_id构造函数参数来改变id属性和<lable>标签的行为：False代表彻底不显示id和label；True的话则会使用变量的属性名（就像上面例子一样）来替换格式化字符串（比如'id_%s'）里的格式化字符。另外标签里结尾的:符号会被label_suffix参数重写，这个参数就是一个简单的字符串。

下面展示了如何创建一个关闭了auto_id和label_suffix（通过把它设为空）的PersonForm实例：

```
pf = PersonForm(auto_id=False, label_suffix='')
```

以及first变量在表单显示时候的样子：

```
<tr><th>First</th><td><input type="text" name="first" maxlength="100" /></td></tr>
```

最后是相同的设置，但是这次为auto_id和label_suffix提供了自定义的字符串值：

```
pf = PersonForm(auto_id='%s_id, label_suffix='?')
```

它的输出如下：

```
<tr><th><label for="first_id">First?</label></th><td><input id="first_id"
type="text" name="first" maxlength="100" /></td></tr>
```

显示全部表单

默认情况下，打印表单用的是as_table方法，它会用<tr>和<td>标签按每行两个变量来打印，为了灵活性它会省略<table>标签。as_table还有一个兄弟as_p方法，它使用的是段落标签，而as_ul则会使用列表项标签（但是会忽略包裹它们自己的标签）。

> **注意**
>
> 表单省略“外围”的包裹标签，比如<table></table>是因为如果包含它们，就会难以将整个表单继承到模板的HTML里去。提交按钮也是一样——很多模板设计都要求使用不同的方法来提交表单，比如<input type="button" />或<input type="submit" />，所以Django把这个留给你自己决定了。

当使用这种方式显示表单的时候，验证错误也是自动打印的，如果发生错误：就会根据所使用输出方法的不同，在相应的变量旁边显示一个标签和若干标签。as_table和as_ul会在和变量自身同一个标签里显示错误列表（分别是用<td>和标签），而as_p则会创建一个新的段落来显示错误列表。不管是哪种情况，错误都会显示在前面的表单元素之前。

你还可以通过继承django.forms.util.ErrorList并将子类作为error_class参数传递给相应的表单来自定义错误列表的显示方式。而且若是想要改变变量/错误列表的显示顺序，只需要重新排列它们在Form类里出现的顺序就可以了——够简单吧。

逐个显示表单

除了刚才介绍的方法，你还可以更细致地控制组织表单。通过表单的字典键可以访问到每一个单独的Field对象，让你可以随时随地显示它们。有了Python的duck typing，你甚至还可以迭代表单本身。不管你通过什么方式获得它们，每一个变量都有自己的errors属性，这是一个

类似列表的对象，字符串表示和之前显示全部方法里的无序列表是一样的（重写的方法也相同）。

在oldforms里，重写一个表单变量默认HTML表示最简单的办法就是访问变量的data属性，然后用自定义的HTML把它包裹起来——这个技巧在newforms里也可以用。不过Widget出现后基本上就削弱了它存在的必要性了。

Widget

在Django表单的眼里，widget就是一个知道如何显示HTML表单元素的对象。Django以相似的风格为模型Field子类和表单Field子类各提供了一个大小适中的默认类库Widget子类。每个表单变量都专门配备了一个Widget，这样在模板里渲染表单的时候，它的数据就知道怎么显示出来。例如，CharField默认使用的是Widget的一个子类TextInput，它会被渲染成<input type="text" />。

通常这些默认的变量widget就已经够用了，你甚至都注意不到它们的存在。不过有时候你会发现需要修改一个变量widget的属性或是把widget整个都换掉。前面这个情况一般更常见一点，它提供了一种让你修改变量的HTML属性的方法。下面这个例子修改了文本变量的“size”属性：

```
from django import newforms as forms

class PersonForm(forms.Form):
    first = forms.CharField(max_length=100, required=True)
    last = forms.CharField(max_length=100, required=True)
    middle = forms.CharField(max_length=100,
        widget=forms.TextInput(attrs={'size': 3}
)
```

使用这个表单得到的middle变量就会变成这样：

```
<input id="id_middle" maxlength="100" type="text" name="middle" size="3" />
```

你可以这么修改是因为Widget子类（比如TextInput）接受了一个attrs字典，它能直接映射到HTML标签的属性上去。在这个例子里，我们不希望限制中间名的实际输入长度（用户还是可以输入长达100个字符），但是我们却希望显示长度比默认的短一些。

重写一个变量的默认widget

Field子类的widget参数可以用来整个替换默认的widget，你只要传递另一个Widget子类就行了，例如，可以用一个Textarea来代替TextInput。利用这一点就可以清楚地分离一个变量的显示（widget）和它的验证行为（表单变量）。这还意味着如果内置的Widget子类不能满足你的需要，你还可以定义自己的Widget子类。

虽然重新定义widget的细节超出了本章的范畴，我们还是可以分享一下如何简单快捷地使用Widget子类来节省时间的方法。如果你经常需要用到某个特定widget的attrs字典的话，你就可以考虑继承这个Widget然后给它定义一个默认的attrs字典。

```
from django import newforms as forms

class LargeTextareaWidget(forms.Textarea):
    def __init__(self, *args, **kwargs):
        kwargs.setdefault('attrs', {}).update({'rows': 40, 'cols': 100})
        super(LargeTextareaWidget, self).__init__(*args, **kwargs)
```

上面例子里对字典耍了一些小聪明。setdefault在给定的键存在的情况下会返回现有的值，若给定的键不存在，那么就返回提供的值。但是，它还修改了字典中来永久保存那个值。它用在这里是确保kwargs关键字参数字典一定拥有attrs字典，不管原来的构造函数参数是什么。然后我们就可以用update来更新attrs字典添加我们需要的默认值了。

最终除了默认拥有40行和100列之外，我们这个新的LargeTextarea widget和普通的Textarea一模一样。然后我们就可以为所有希望显示的比普通文本区域大一点的变量应用这个新的widget了。下面这个例子里，假设我们把和表单相关的自定义类保存在一个本地应用forms.py里。

forms.py.

```
from django import newforms as forms
from myproject.myapp.forms import LargeTextareaWidget

class ContentForm(forms.Form):
    name = forms.CharField()
    markup = forms.ChoiceField(choices=[
        ('markdown', 'Markdown'),
        ('textile', 'Textile')
    ])
    text = forms.Textarea(widget=LargeTextareaWidget)
```

当然你还可以更进一步。因为Field子类的widget参数只是用来设置它的widget属性，所以我们可以继承这个变量本身来让它总是使用我们的自定义widget。

```
class LargeTextareaWidget(forms.Textarea):
    def __init__(self, *args, **kwargs):
        kwargs.setdefault('attrs', {}).update({'rows': 40, 'cols': 100})
        super(LargeTextareaWidget, self).__init__(*args, **kwargs)

class LargeTextarea(forms.Field):
    widget = LargeTextareaWidget
```

现在我们就可以修改之前的创建表单示例来使用这个自定义的变量了。

```
class ContentForm(forms.Form):
    name = forms.CharField()
    markup = forms.ChoiceField(choices=[
        ('markdown', 'Markdown'),
        ('textile', 'Textile')
    ])
    text = LargeTextarea()
```

不要忘了，Django就是纯Python。这代表在有需要的时候，你很容易就能像这样把各种类

和对象替换出去。记住这一点有助于你发现其他可以利用自定义的地方。

6.3 总结

在这一章里，你学习了Django的模板语法以及如何用context字典来渲染模板，还包括了更复杂的主题诸如模板继承和包含等。此外，你知道了怎样生成表单（单独的或是表示某个特定模型类的）以及验证它们的数据和显示为HTML。最后，我们展示了一些通过widget自定义表单的方法。

这一章是第二部分的结尾，现在你对Django提供的功能应该已经有了一个相当完整的了解。从模型定义，URL和请求处理，到现在的模板和表单。在下面第三部分的4章里，将要展示使用了这些你已经看到的知识所构建起来的应用示例，并在其中穿插一些新的或是扩展的概念。

第三部分 Django应用实例

第7章 Photo Gallery
第8章 内容管理系统
第9章 Liveblog
第10章 Pastebin

第7章 Photo Gallery

很多由内容驱动的网站都有一个常见的特点就是它们不仅允许用户添加文本，同时还可以添加文件——Office文档、视频、PDF文件，当然还有到处都是的图片。在这个应用示例里，我们要向你展示如何在一个简单的Gallery类型的应用程序里使用Django图片上传变量ImageField。此外，我们还要创建一个自定义的ImageField子类来自动生成图片预览。最后，我们还为之设计一个动态的根URL以尽可能地方便部署。

我们要构建的这个应用程序已经经过简化了，一组通用的Item，每一个都可以拥有多个Photo，这个层次表示为一个Django项目gallery及其内部的一个items项目（取不出更好的名字了）。

你可以扩展这个应用程序来构建一个更常见的图片站点，Item可以变得更像是一个容器或者文件夹，专门用来组织照片。又或者你可以把它做成类似陈列馆的一个应用，每个Item可以拥有额外的属性（比如汽车型号、厂商和年份）来适应这种复杂度。对于我们这个例子，你可以把Item想象成一本独立的相册。

我们不打算做任何不必要的工作，所以应用程序会尽量使用通用视图，并且所有数据输入都是经由Django的内建admin来完成。这样，它的架构就相当紧凑了：

- 首页上静态的模板欢迎信息。
- 首页显示了一个“陈列柜”（一小组图片预览）。
- 列表页面显示所有站点上的Item。
- 每个Item的细节视图都列出所有属于它的Photo（也是预览）。
- 每个Photo的细节视图会全尺寸显示图片。

我们从定义模型开始，然后一步步地让文件能从admin应用里上传。接着详细讨论创建自定义模型变量。最后，我们看如何将DRY原则应用到URL上以及创建用来向全世界展示预览和图片的前台模板。

> **注意**
>
> 这个例子假设你用的是Apache+mod_python，当然你可以修改它以适应其他部署策略。因为一个Gallery需要服务大量的静态内容（比如图片），Django的开发服务器不太适合它。你可以在附录B里找到更多关于Apache配置的信息。

7.1 模型

下面是本应用的models.py，除了稍后要做出的一个小改动外，这就是一个完整的文件了。注意get_absolute_url方法使用了@permalink装饰器，在本章结束之前会介绍到它。

```
class Item(models.Model):
    name = models.CharField(max_length=250)
    description = models.TextField()

    class Meta:
        ordering = ['name']

    def __unicode__(self):
        return self.name

    @permalink
    def get_absolute_url(self):
        return ('item_detail', None, {'object_id': self.id})

class Photo(models.Model):
    item = models.ForeignKey(Item)
    title = models.CharField(max_length=100)
    image = models.ImageField(upload_to='photos')
    caption = models.CharField(max_length=250, blank=True)

    class Meta:
        ordering = ['title']

    def __unicode__(self):
        return self.title

    @permalink
    def get_absolute_url(self):
        return ('photo_detail', None, {'object_id': self.id})

class PhotoInline(admin.StackedInline):
    model = Photo
class ItemAdmin(admin.ModelAdmin):
    inlines = [PhotoInline]

admin.site.register(Item, ItemAdmin)
admin.site.register(Photo)
```

这里，Item非常简单，而Photo才是真正的主角——它不仅和父亲Item有关联，而且还有标题、图片文件本身和一个可选的说明。这两个对象都向admin应用注册了自己，它们也都有一个Meta.ordering属性集。

7.2 准备文件上传

在可以上传文件到图片网站之前，我们需要告诉Django要把它们放到哪里去。FileField和ImageField会将上传的数据保存在settings.py里定义的MEDIA_ROOT中的一个子目录下，它会在upload_to变量参数里定义。在我们的模型代码里，我们已经把它设置成了'photo'，所以若是settings.py里是这么设置的话：

```
MEDIA_ROOT = '/var/www/gallery/media/'
```

那么最终照片就会被保存在 /var/www/gallery/media/photos/ 里。如果这个目录不存在的话，就要先创建它，另外Web服务器运行的用户或者用户组需要有对这个目录的写权限。在我们的Debian系统上，Apache是以www-data用户身份运行的，所以我们要像下面这样设置一番（命令行的使用细节请参考附录A）：

```
user@example:~ $ cd /var/www/gallery/media
user@example:/var/www/gallery/media $ ls
admin
user@example:/var/www/gallery/media $ mkdir photos
user@example:/var/www/gallery/media $ ls -l
total 4
lrwxrwxrwx 1 root  root    59 2008-03-26 21:41 admin ->
/usr/lib/python2.4/site-packages/django/contrib/admin/media
drwxrwxr-x 2 user user 4096 2008-03-26 21:44 photos
user@example:/var/www/gallery/media $ chgrp www-data photos
user@example:/var/www/gallery/media $ chmod g+w photos
user@example:/var/www/gallery/media $ ls -l
total 4
lrwxrwxrwx 1 root  root       59 2008-03-26 21:41 admin ->
/usr/lib/python2.4/site-packages/django/contrib/admin/media
drwxrwxr-x 2 user www-data 4096 2008-03-26 21:44 photos
```

上面这种情况只有在你的普通用户也是属于www-data用户组的时候才适用——根据系统设置的不同，你或许会需要用sudo等类似方法才能设置完成。我们发现在Web服务器域上执行很多系统任务的时候，把自己设置成它用户组的一部分会很方便。只要目录或文件有小组写权限（就像上面的例子），Web服务器和我们的用户就都能访问它们。

最后，通常更完整的应用程序会需要在media目录里建立另外一些符号链接（毕竟你还需要CSS和JavaScript），你的Web服务器需要能正确设置来适应它们。例如，假设你用的是Apache加mod_python，那么你需要在Django把持的URL空间上留出一些空隙，以便Apache能够直接处理你的媒体文件。更多mod_python配置的信息请参见附录B。

7.3 安装PIL

到此（当我们把自定义应用加到settings.py里并且运行syncdb后），我们几乎就准备好上传图片了。不过就像马上要看到的，我们还差一口气。要是现在我们就加载admin站点（安装好Django项目并且设置好照片上传文件夹），基本上我们会看到图7.1这样的画面。

也就是说，我们需要安装一个特殊的Python类库PIL（Python Imaging Library）才能使用ImageField。PIL是一个常用的Python类库，它能处理各种各样的图片格式。ImageField用它来验证上传的文件确实是图片，并且若是你使用可选的height_field和width_field的话，还会把它们的长和宽保存下来。我们还要用它来生成图片预览。

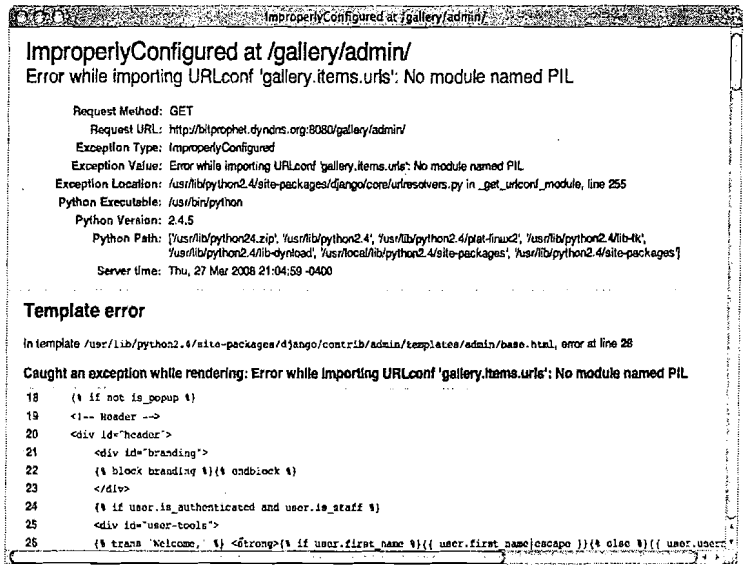


图7.1 没有安装PIL就会看到这个

要在基于UNIX的系统上（比如Linux或Mac）安装PIL，从http://www.pythonware.com/products/pil下载源码包，解压后在install目录里执行setup.py install即可。在Win32上，则可以下载适用于你Python版本的.exe二进制文件来安装。

不管在什么平台上，安装第三方Python软件包最简单的方法就是Easy Install。它知道怎么处理依赖关系，并且把下载和安装合并到一步完成。你只要运行easy_install pil就可以完成刚刚描述的所有步骤。更多关于获取和使用Easy Install工具的信息，请访问PEAK开发者中心http://peak.telecommunity.com/DevCenter/EasyInstall。

7.4 测试ImageField

PIL安装完成并且重启（或者重新载入）Web服务器后，PIL的错误就应该消失了，我们就可以继续来简单测试一下ImageField。

由于gallery模型组织的方式，我们不可以不关联一个Item就随便上传一张图片。仔细检查

模型的定义后我们发现admin及其内联类都设置好了，所以我们的Photo对象可以随着它们的父亲Item一起编辑。这样我们就可以轻松地在定义item的同时添加图片了，如图7.2所示。

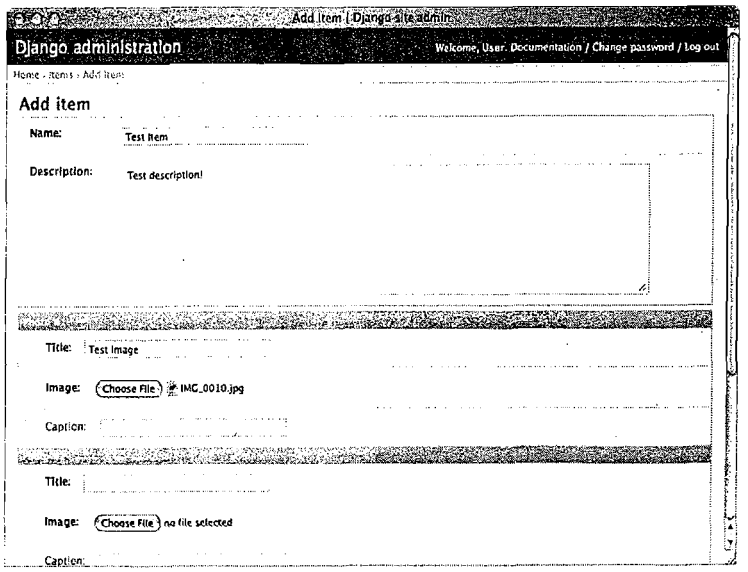


图7.2 ImageField

保存新的Item之后，所选择的图片（在这里是一幅我们一位作者的宠物兔的照片）也会随之上传并保存。我们可以通过admin来验证这一点，如图7.3所示。

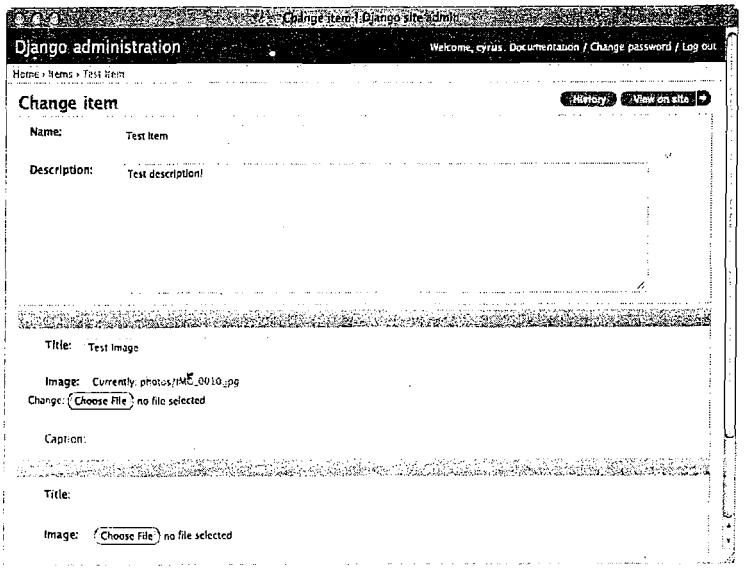


图7.3 文件上传后的admin界面

注意第一个Photo里文件选择器上面的Currently:link——点击它就会显示上传的图片，如图7.4所示。



图7.4 检查当前ImageField的值

我们也可以在命令行上验证上传的文件。

```
user@example:/var/www/gallery/media/photos $ ls -1
total 144
-rw-r -r - 1 www-data www-data 140910 2008-03-27 21:26 IMG_0010.jpg
```

虽然我们花费了一些篇幅来解释整个过程，不过你可以看到，其实这个设置和运行图片上传的过程非常直观——到现在为止，我们真正做的也就是定义一个模型，创建一个保存图片的文件夹，还有安装图片类库。现在我们终于要向你展示一点有趣的东西了，即如何扩展ImageFicld来生成图片预览。

7.5 构建自定义File变量

由于Django并没有提供生成预览的功能，所以我们准备通过继承ImageField来编写自己的模型变量，让它能无缝地处理图片预览的生成、删除和显示。Django的官方文档为如何重新编写整个模型变量提供了出色的信息——不过在这里，我们只要调整一下现有的行为就可以了，这更简单也更常见。

不要害怕源码

程序员经常会把他们使用的类库（哪怕是开源的）当成是定义了输入输出行为的黑盒来使用，把它们看的很神秘。虽然有时候这么做是正确的，比如在冗长的低级语言里，这些库可能真的是又巨大又宏伟，但是对Python源码来说这种情况却不常见。

写得好的Pythonic库通常都是很容易理解和修改的，Django也不例外。我们不会假装整个代码库都已经完美重构和注释了，但是大部分都是相当健壮的，程序员（哪怕是初学乍

练）也可以通过深入代码了解Django工作机理来获取很多益处。这一章里我们的工作就没有完整的文档——但是通过阅读django.db.models.ImageField及其父类的源码还是能比较容易地搞清楚它们的。

要完成这个功能，我们需要重写ImageField父类里的4个方法，并增加一个简单的私有方法来帮助重构。这一章里的代码已经有完整的注释（文档在理解新领域的时候是非常有帮助的），但是这里为了阅读方便，我们把它们几乎都删掉了。

我们把预览ImageField（很自然地）起名为ThumbnailImageField，将它保存在gallery.items.fields之下。它仅由一些import，一个重构的工具函数，以及一组修改了一些Django内置类的子类。如果你对Python面对对象的继承方式还不熟悉的话，请参阅第1章，“实战Django Python”。

我们自上而下地来解释一下这个文件。

初始化

每个Python文件（几乎没有例外的）都由import开始，这个文件也是一样。

```
from django.db.models.fields.files import ImageField, ImageFieldFile
from PIL import Image
import os
```

Import还是很简单的，这里我们需要的就是父类ImageField和ImageFieldFile，PIL里用来生成预览的Image类，还有负责处理预览文件的内置os模块。

```
def _add_thumb(s):
    """
    Modifies a string (filename, URL) containing an image filename, to insert
    '.thumb' before the file extension (which is changed to be '.jpg').
    """
    parts = s.split(".")
    parts.insert(-1, "thumb")
    if parts[-1].lower() not in ['jpeg', 'jpg']:
        parts[-1] = 'jpg'
    return ".".join(parts)
```

_add_thumb是一个工具函数，正如docstring里描述的那样（总是要好好利用docstring），它接受原图像文件的路径，然后插入字符串“.thumb”。所以rabbit.jpg上传后得到的预览文件名就是rabbit.thumb.jpg。由于我们的代码只能生成JPEG的预览，所以如有需要，它还会修改文件的扩展名。

```
class ThumbnailImageField(ImageField):
    """
    Behaves like a regular ImageField, but stores an extra (JPEG) thumbnail
    image, providing get_FIELD_thumb_url() and get_FIELD_thumb_filename().
```

```
    Accepts two additional, optional arguments: thumb_width and thumb_height,
    both defaulting to 128 (pixels). Resizing will preserve aspect ratio while
    staying inside the requested dimensions; see PIL's Image.thumbnail()
    method documentation for details.
    """
    attr_class = ThumbnailImageFieldFile
```

这个类就很简单了——我们从ImageField继承了一个新的子类，然后写下一大串docstring。这样，任何人在使用Python的帮助系统，或是自动化文档工具时，都能很容易理解它的作用。

所以实际上这个类只有一行代码，attr_class是用来更新一个特殊的类，我们的变量用这个类来访问属性。在下一节里我们会详细讨论它。初始化介绍的最后一部分就是__init__：

```
def __init__(self, thumb_width=128, thumb_height=128, *args, **kwargs):
    self.thumb_width = thumb_width
    self.thumb_height = thumb_height
    super(ThumbnailImageField, self).__init__(*args, **kwargs)
```

我们重写的__init__相当简单，只需要把预览文件在待会改变大小时的最大长和宽保存下来就行了。在需要不同预览大小的时候，重用它就很容易了。

给变量添加属性

大多数变量都是相对自我约束的，不会去随便改动包含它们的模型对象，但是在我们这个例子里，我们希望能够方便地得到我们提供的额外信息（预览文件的文件名和URL）。方法是去继承ImageField用来管理属性的一个特殊的类，ImageFieldFile。ImageField会用它去查找自身的变量。

例如，用myobject.image.path就可以得到一个叫image的ImageField对象在文件系统上的路径。这里的.path就是ImageFieldFile的一个属性。由于Django会尽量缓存文件数据而把具体文件代理到更低的层次上去，所以这是通过Python的properties完成的。（你可以参阅第1章来复习一下内置函数property的内容。）

下面的代码展示了Django是怎么实现ImageFieldFile.path的：

```
ImageFieldFile.path:

    def _get_path(self):
        self._require_file()
                return self.storage.path(self.name)
    path = property(_get_path)
```

这段代码取自FieldFile类（它是ImageFieldFile的父类）。回想一下前面的帮助函数_add_thumb是怎么转换一个给定文件路径的，你就能猜到我们要怎么把.thumb_path和.thumb_url属性加到变量里去了：

```
class ThumbnailImageFieldFile(ImageFieldFile):
    def _get_thumb_path(self):
        return _add_thumb(self.path)
    thumb_path = property(_get_thumb_path)
```

```
    def _get_thumb_url(self):
        return _add_thumb(self.url)
    thumb_url = property(_get_thumb_url)
```

因为.path和.url的getter函数已经定义好了，而且它们会去安全地处理那些重复性的操作（比如上面_get_path代码里的self._require_file调用），所以我们可以省略这些代码。我们只需要调用_add_thumb转换并且把结果用property函数附加到需要的属性名上即可。

当ThumbnailImageFieldFile在ThumbnailImageField上定义以及在ThumbnailImageField里由class引用之后，我们给变量添加了两个新的属性，你可以在Python代码或是模板里使用它们：myobject.image.thumb_path和myobject.image.thumb_url（这里myobject是Django的模型实例，image是那个模型上的ThumbnailImageField对象）。

继承ImageFieldFile然后将子类和ImageField的子类连结起来可能不是很常见的手法，绝大多数自定义的模型变量甚至根本用不着深入到这个地步。事实上，作为Django的用户，你可能永远见不到模型里的这一面。然而它展示了Django核心团队不仅仅在公共API上尽力保持扩展性，而且在框架内部也是如此。

现在我们已经可以访问所需的预览文件的URL和文件系统路径了，接下来就是实际创建（和删除）这个预览文件。

保存和删除预览文件

创建预览文件的关键之处，就是ThumbnailImageFieldFile（不是ThumbnailImageField！）里的save方法：

```
def save(self, name, content, save=True):
    super(ThumbnailImageFieldFile, self).save(name, content, save)
    img = Image.open(self.path)
    img.thumbnail(
        (self.field.thumb_width, self.field.thumb_height),
        Image.ANTIALIAS
    )
    img.save(self.thumb_path, 'JPEG')
```

调用父类的save方法会帮你正常地处理主要图片文件的保存操作，所以我们的方法只需要做三件事情，即打开原图片文件，创建预览，将预览保存为预览文件名。注意这里的self.field让我们能访问属于File对象的Field，即保存了预览图片大小的地方。利用一开始导入的PIL Image类，这里的代码其实非常简单。

最后，当删除原图片时，它们的预览文件也要随之一起删除：

```
def delete(self, save=True):
    if os.path.exists(self.thumb_path):
        os.remove(self.thumb_path)
    super(ThumbnailImageFieldFile, self).delete(save)
```

多亏了Python漂亮的语法，这段代码几乎就是在描述自己了。首先获取预览文件名，删除

它（如果文件存在的话，即使不存在，那也算不上什么错误），然后告诉父类去删除它自己的文件（即原图）。如果上面的代码还不够清楚，这里的意思是delete方法会和save一样，当它的容器模型对象被删除时，就会触发ImageField去调用它。

> **操作的顺序**
>
> delete方法里操作的顺序很有讲究，你在继承的时候一定要先考虑清楚。如果先调用super，那么self.thumb_path调用就会出错，因为它会先去调用self.path（回忆一下之前的代码）来保证变量的主文件确实存在！所以我们需要等到最后一刻才能删除主文件，以免我们的类失灵。

7.6 使用ThumbnailImageField

定义完ImageField子类后，我们来看看怎么用它。首先在models.py里加入一个新的import：

```
from gallery.items.fields import ThumbnailImageField
```

然后把Photo模型里的models.ImageField换成带预览的版本：

```
class Photo(models.Model):
    item = models.ForeignKey(Item)
    title = models.CharField(max_length=100)
    image = ThumbnailImageField(upload_to='photos')
    caption = models.CharField(max_length=250, blank=True)
```

重新加载Web服务器后admin里还看不到什么显著的变化，因为我们还没有修改任何与表单有关的变量，如图7.5所示。

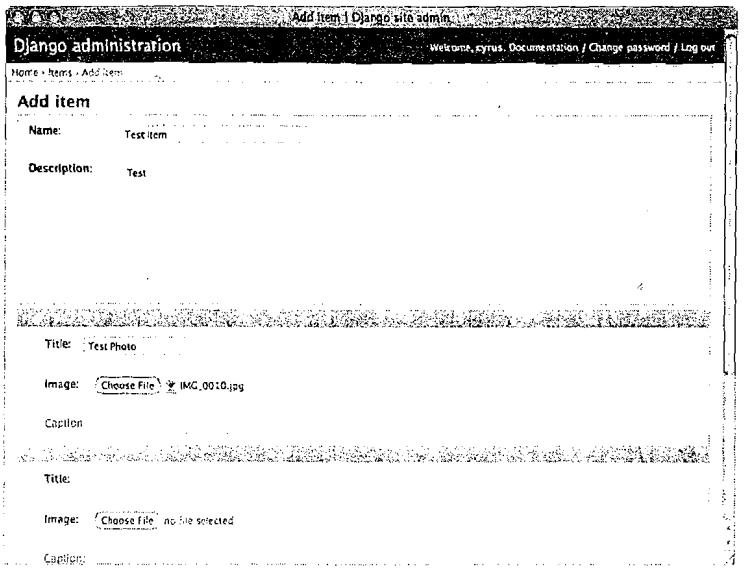


图7.5　和前面没什么两样

即使是在文件上传后，看起来也和之前的例子一模一样，如图7.6所示。

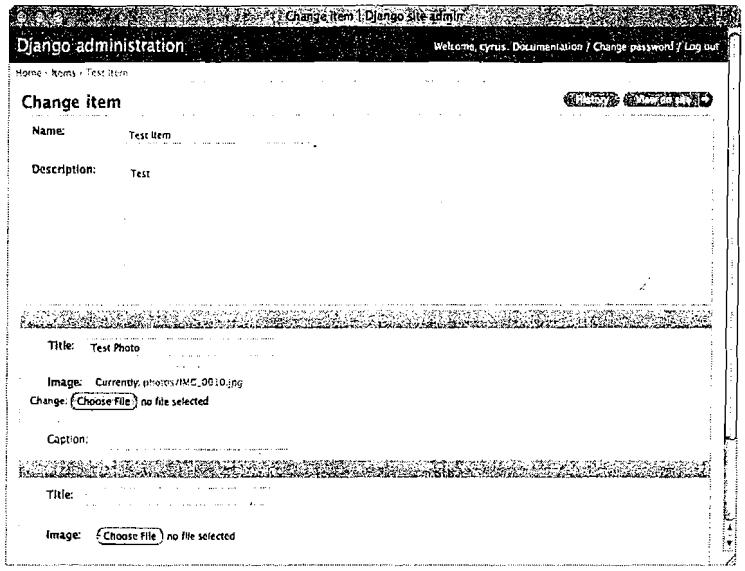


图7.6 和之前的页面还是一样

但是检查一下我们的上传目录，你就能发现我们的劳动果实啦。

```
user@example:/var/www/gallery/media/photos $ ls -l
total 148
-rw-r -r - 1 www-data www-data 140910 2008-03-30 22:15 IMG_0010.jpg
-rw-r -r - 1 www-data www-data   1823 2008-03-30 22:15 IMG_0010.thumb.jpg
```

大功告成！不过还差一点才能看到这个预览，因为我们还没有向你展示应用程序里负责显示它们的模板。在这之前，我们先赶快看一下应用程序里一个比较次要的方面：怎么设置一个简单的方式才好保持URL的DRY呢？

7.7 设置DRY URL

到这里为止，我们关注的只是Gallery应用里模型（model）的部分。现在我们来看看URL的结构，它为理解下一节里的模板打下了必要的基础。不过首先我们需要一点背景知识才好理解这个不太寻常的设置方式。由于这个应用开发的方式，我们希望它无论是在一个域名的顶层（比如http://www.example.com/）还是作为其中一部分（例如http://www.example.com/gallery/）都可以正常工作。

默认情况下，一个Django站点都假设是前面的那种情况——即URL是从域名的根目录开始解析，哪怕Web服务器处理器被挂接在更高的层次上面。因此，一般站点的URL都必须包含整个URL路径，所以一个在 /gallery/ 上的站点里，它需要在所有URL上加上这个字符串前缀[⊖]。

⊖ Django 从1.0起引入了一个新的Apache配置说明，PythonOption django.root <root>，它基本取代了我们在这里介绍的ROOT_URL功能。不过，我们还是把这部分内容保留下来，作为Django“就是Python”的例子之一，以及展示这种方式如何让你可以通过多种方法改变这种行为。

所以我们这里就依照规矩来，把这个值保存在settings.py里的一个变量里，然后在必要的地方引用它。

```
ROOT_URL = '/gallery/'
```

因为还有其他一些依赖于URL路径的settings.py变量，所以我们这里就把它们放在一起了，这里有登录的URL，媒体文件的URL，以及admin的媒体文件前缀。

```
LOGIN_URL = ROOT_URL + 'login/'
MEDIA_URL = ROOT_URL + 'media/'
ADMIN_MEDIA_PREFIX = MEDIA_URL + 'admin/'
```

接下来，又由于Django的URL包含系统的工作方式，我们得用到一个双文件形式的根URLconf设置，这里“普通的”顶层urls.py只用ROOT_URL，然后去调用“真正的”urls.py，而这个文件对ROOT_URL及其相关的内容一无所知。这里是根urls.py：

```
from django.conf.urls.defaults import *
from gallery.settings import ROOT_URL

urlpatterns = patterns('',
    url(r'^%s' % ROOT_URL[1:], include('gallery.real_urls')),
)
```

注意

我们要对ROOT_URL切片来去掉开头的斜杠，因为用到它的settings.py变量（比如LOGIN_URL）为了成为正确的绝对路径的形式会要求自带一个开头的斜杠。但是由于Django的URL解析会忽略那个开头的斜杠，所以这里必须要去掉它才能让我们的URL正确解析。

这里是我们“真正的”的根URLconf，称之为real_urls.py（想不出更好的名字了）：

```
from django.conf.urls.defaults import *
from django.contrib import admin

urlpatterns = patterns('',
    url(r'^admin/(.*)', admin.site.root),
    url(r'^', include('gallery.items.urls')),
)
```

最后，要是模板也能访问ROOT_URL的话就能方便地构建出符合DRY原则的includes URL，比如那些CSS或是JavaScript需要的includes。这只需一个小小的context处理器就能做到（这个请参阅前面第6章）。

```
from gallery.settings import ROOT_URL


def root_url_processor(request):
    return {'ROOT_URL': ROOT_URL}
```

完成了！这个Django项目经过一系列的调整之后，所有一切就都围绕着ROOT_URL的值

了——现在它的值是'/gallery/'，表示应用程序的位置是http://www.example.com/gallery/。要是你想把它部署到http://www.example.com/ 上去的话，只需要把ROOT_URL改成'/'（并且更新Web服务器的设置把Django挂接到根目录层上去）就行了。

7.8 Item应用的URL布局

为了完成URL结构的DRY“属性”，我们要对我们对象里的get_absolute_url方法应用三个步骤。第一个也是最重要的部分就是（你在本书里已经见过的）用url函数来定义我们的URLconf，这允许我们给URL取一个唯一的名字。下面是包含在items应用里的urls.py：

```
from django.conf.urls.defaults import *
from gallery.items.models import Item, Photo

urlpatterns = patterns('django.views.generic',
    url(r'^$', 'simple.direct_to_template',
        kwargs={
            'template': 'index.html',
            'extra_context': {'item_list': lambda: Item.objects.all()}
        },
        name='index'
    ),
    url(r'^items/$', 'list_detail.object_list',
        kwargs={
            'queryset': Item.objects.all(),
            'template_name': 'items_list.html',
            'allow_empty': True
        },
        name='item_list'
    ),
    url(r'^items/(?P<object_id>\d+)/$', 'list_detail.object_detail',
        kwargs={
            'queryset': Item.objects.all(),
            'template_name': 'items_detail.html'
        },
        name='item_detail'
    ),
    url(r'^photos/(?P<object_id>\d+)/$', 'list_detail.object_detail',
        kwargs={
            'queryset': Photo.objects.all(),
            'template_name': 'photos_detail.html'
        },
        name='photo_detail'
    ),
)
```

如你所见，这个应用程序包含了一个首页，一个所有相册的列表，相册页面，以及相片页面，每一个都有直观的名字。在下一节的模板里，我们将通过{% url %}模板标签来引用这些名字，此外，包裹get_absolute_url的permalink装饰器也要引用它们，像这样：

```
class Item(models.Model):
    name = models.CharField(max_length=250)
    description = models.TextField()

    class Meta:
        ordering = ['name']

    def __unicode__(self):
        return self.name

    @permalink
    def get_absolute_url(self):
        return ('item_detail', None, {'object_id': self.id})
```

permalink装饰器要求它包裹的函数返回一个三元组（URL名、一列位置参数和一个命名参数的字典）来重新组成URL。在上面的例子里，item_detail视图不接受位置参数，但是接受了一个命名参数，而那个参数正是我们在get_absolute_url里提供的。

这种设置方式即便在URL结构发生变化的时候也可以让Item.get_absolute_url能返回适合的URL，从而达到保持DRY的目的（这里也不是没有代价，比如要是去掉装饰器的话，get_absolute_url的行为就会显得很怪异等）。

7.9 用模板把它们都串在一起

完成自定义模型变量以及调整好URL设置之后，最后剩下的（因为这里只用到了通用视图）就是模板了。我们采用的是一种简单的继承方式来最大化DRY，首先是基本的结构模板和一些CSS。

```
<html>
    <head>
        <title>Gallery - {% block title %}{% endblock %}</title>
        <style type="text/css">
            body { margin: 30px; font-family: sans-serif; background: #fff; }
            h1 { background: #ccf; padding: 20px; }
            h2 { background: #ddf; padding: 10px 20px; }
            h3 { background: #eef; padding: 5px 20px; }
            table { width: 100%; }
            table th { text-align: left; }
        </style>
    </head>
    <body>
        <h1>Gallery</h1>
        {% block content %}{% endblock %}
    </body>
</html>
```

接下来是首页。本章的这个例子应用了一点轻型的类似CMS的功能，在admin里显示了一堆“欢迎”用语，这里为简单起见就省略了。我们在这里用的是一段静态的欢迎标语和一列三

个由URLconf控制的特色Item。（它现在显示的是全部内容，不过很容易改成其他形式。）

```
{% extends "base.html" %}

{% block title %}Home{% endblock %}
{% block content %}

<h2>Welcome to the Gallery!</h3>
<p>Here you find pictures of various items. Below are some highlighted
items; use the link at the bottom to see the full listing.</p>

<h3>Showcase</h3>
<table>
    <tr>
    {% for item in item_list|slice:":3" %}
        <td>
            <a href="{{ item.get_absolute_url }}"><b>{{ item.name }}</b><br />
            {% if item.photo_set.count %}
                <img src="{{ item.photo_set.all.0.image.thumb_url }}" />
            {% else %}
                <span>No photos (yet)</span>
            {% endif %}
            </a>
        </td>
    {% endfor %}
    </tr>
</table>
<p><a href="{% url item_list %}">View the full list &raquo;</a></p>

{% endblock %}
```

上面的模板代码渲染出来的页面视图如图7.7所示。

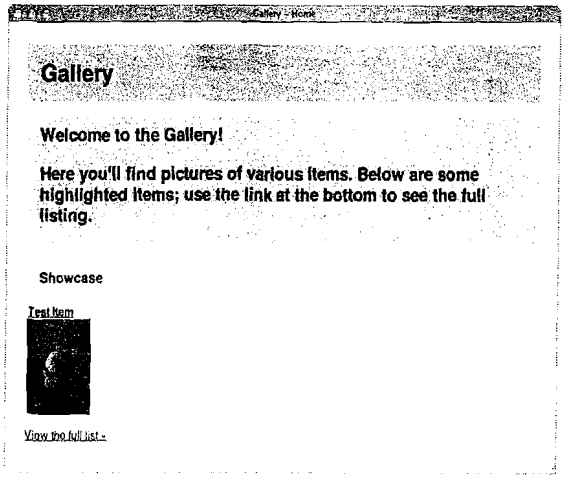


图7.7 Gallery的首页

注意这里分别通过get_absolute_url和{% url %}链接到图片细节页面和相册列表的用法，还

有每个相册列表上第一幅图片的image.thumb_url。这个“用哪张预览作为封面”的问题可以通过更新Photo模型来选用一张特定的图片作为“代表”来改进——这是应用程序很多可扩展的地方之一。

相册列表（items_listing）是一个比首页上的特色列表更完整的版本，它们用的手法其实是一样的，如图7.8所示。

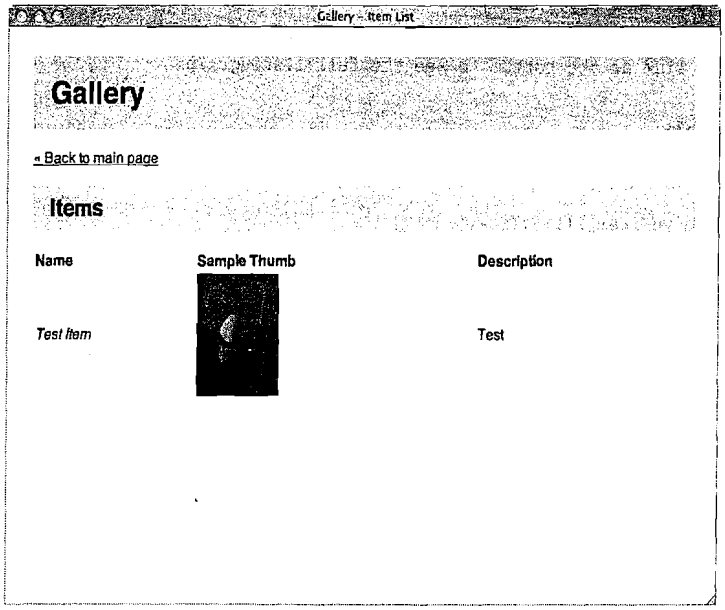


图7.8　Gallery列表页面

```
{% extends "base.html" %}

{% block title %}Item List{% endblock %}

{% block content %}
<p><a href="{% url index %}">&laquo; Back to main page</a></p>

<h2>Items</h2>
{% if object_list %}
<table>
    <tr>
        <th>Name</th>
        <th>Sample Thumb</th>
        <th>Description</th>
    </tr>
    {% for item in object_list %}
    <tr>
        <td><i>{{ item.name }}</i></td>
        <td>
            {% if item.photo_set.count %}
            <a href="{{ item.get_absolute_url }}">
```

```
            <img src="{{ item.photo_set.all.0.image.thumb_url }}" />
        </a>
        {% else %}
        (No photos currently uploaded)
        {% endif %}
    </td>
    <td>{{ item.description }}</td>
    </tr>
    {% endfor %}
</table>
{% else %}
<p>There are currently no items to display.</p>
{% endif %}

{% endblock %}
```

相册的细节视图（items_detail.html）和相册列表视图差不多，就是它会列出所有照片而不只是封面，如图7.9所示。

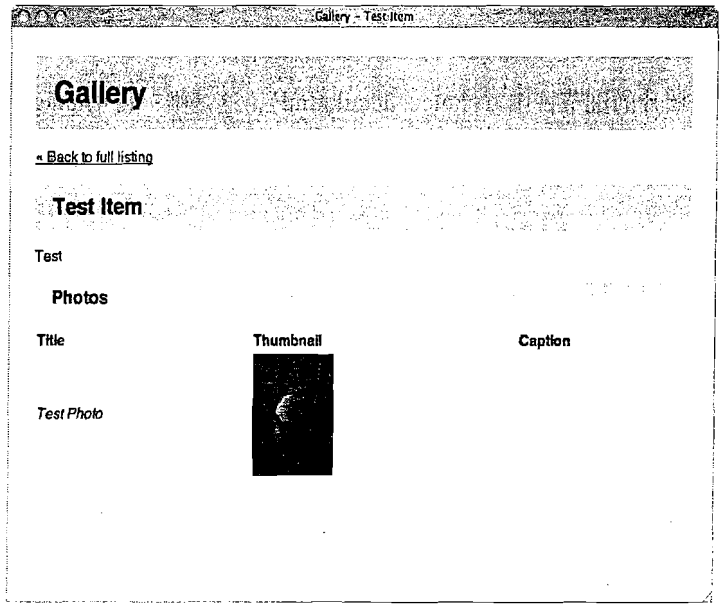


图7.9 相册细节页面

```
{% extends "base.html" %}

{% block title %}{{ object.name }}{% endblock %}

{% block content %}

<p><a href="{% url item_list %}">&laquo; Back to full listing</a></p>
<h2>{{ object.name }}</h2>
<p>{{ object.description }}</p>
```

```
<h3>Photos</h3>
<table>
    <tr>
        <th>Title</th>
        <th>Thumbnail</th>
        <th>Caption</th>
    </tr>
    {% for photo in object.photo_set.all %}
    <tr>
        <td><i>{{ photo.title }}</i></td>
        <td>
            <a href="{{ photo.get_absolute_url }}">
                <img src="{{ photo.get_image_thumb_url }}" />
            </a>
        </td>
        <td>{{ photo.caption }}</td>
    </tr>
    {% endfor %}
</table>

{% endblock %}
```

最后即是每张照片的细节视图（photos_detail.html），这是唯一使用image.url的地方，如图7.10所示。

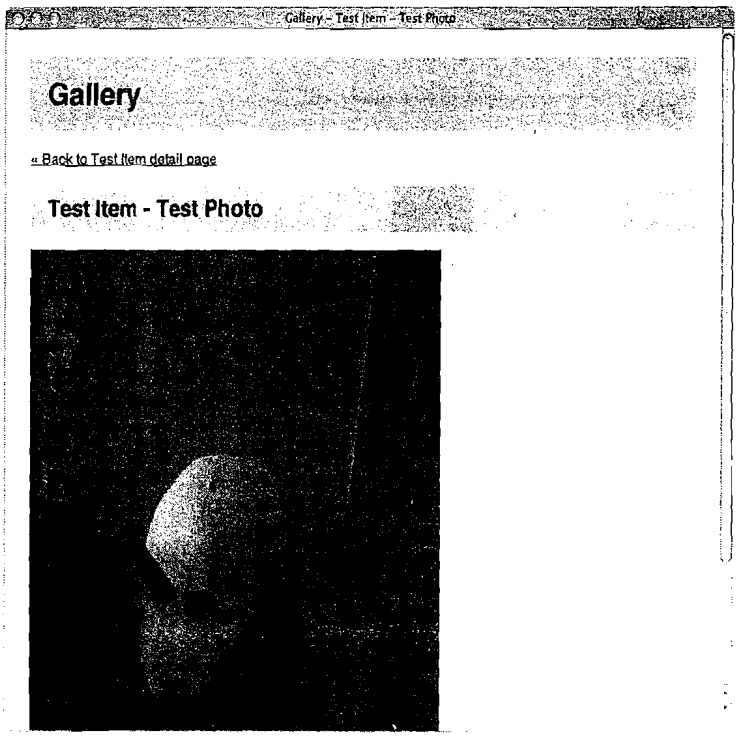


图7.10　图片细节视图

```
{% extends "base.html" %}

{% block title %}{{ object.item.name }} - {{ object.title }}{% endblock %}

{% block content %}

<a href="{{ object.item.get_absolute_url }}">&laquo; Back to {{
object.item.name }} detail page</a>

<h2>{{ object.item.name }} - {{ object.title }}</h2>
<img src="{{ object.get_image_url }}" />
{% if object.caption %}<p>{{ object.caption }}</p>{% endif %}

{% endblock %}
```

7.10 总结

在风风火火地介绍完后，希望你对整个项目是怎么一同工作的有了一个比较完整的概念。

- 我们定义了模型，然后用admin来展示如何上传图片，包括必要的系统设置。
- 因为需要图片预览的功能，所以定义了一个Django图片变量的子类及其相关的文件类，这里我们重写了几个方法来改变图片的大小，以及提供对预览的访问。
- 我们完全利用了Django的DRY URL特性，包括实现一个“根URL”设置（就像那个在1.0之前刚刚加到Django核心里的一样）来帮助我们保持URL灵活。
- 最后，我们创建了一些简单的模板让用户可以浏览和查看图片。

新变量或是模型方法。

不过若这些对你都不是问题的话，Flatpages还是相当有用的。我们在下面几节里会探讨如何设置和使用Flatpages，完成一个更强壮的自定义CMS应用。

激活Flatpages应用

以下是让Flatpages能跑起来的几个必要步骤：

1. 用django-admin.py创建一个新的Django项目。

2. 打开项目的settings.py并更新MIDDLEWARE_CLASSES设置，添加django.contrib.flatpages.middleware。

3. 把django.contrib.flatpages和django.contrib.admin加入到settings.py里的INSTALLED_APPS变量里。

4. 运行manage.py syncdb让Django创建必要的数据表。

5. 更新urls.py，激活默认的admin。

6. （重新）启动你选用的Web服务器。

完成上述步骤后，登录admin站点，你应该能看到图8.1这样的页面。

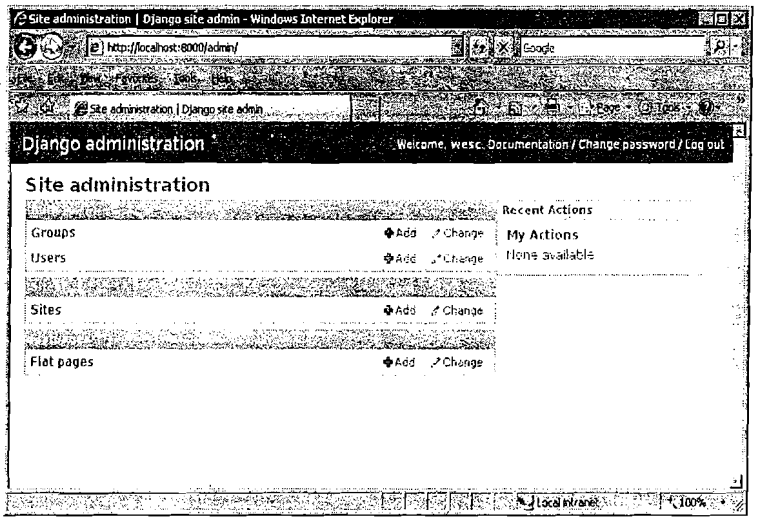


图8.1 激活Flatpages应用后登录admin页面界

点击Add来创建一个新的Flatpage对象（如图8.2所示）。你只需输入url、title和content。确保这里的URL用斜杠开头。

现在创建一到两个页面，这样在测试的时候你就有内容可看了。

下面我们可以直接去看模板的介绍，因为这里除了刚刚输入的URL，你不需要再对它做任何的管理。而且，它会利用一块特殊的Django中间件去拦截404错误，并在Flatpage对象列表里查找请求的URL。如果找到的话，Flatpages应用就会出来接手，要是没找到，那么404就会转去正常的错误处理。

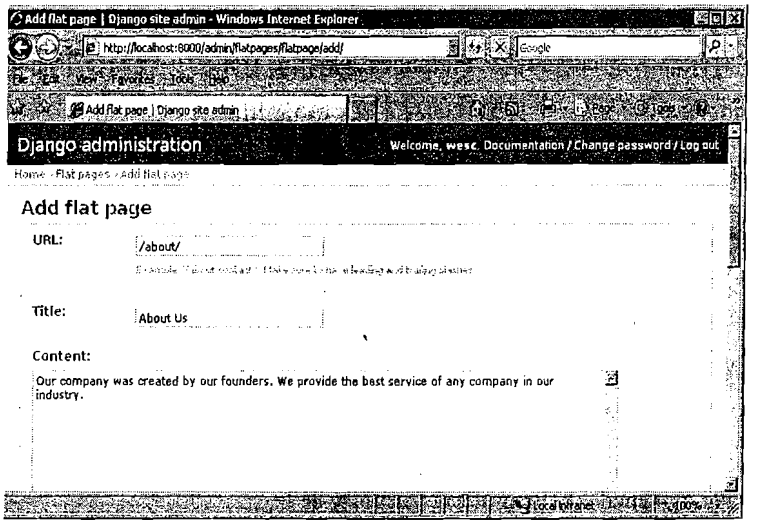


图8.2　Flatpage的Add界面

> **注意**
>
> Flatpages的这种404处理方式意味着你可以把它和正常的Django应用一起使用，就是说你可以简单轻松地指定你的flatpages（“关于”或者“法律”等页面）而不用不断地更新URLconf。

Flatpage模板

每个Flatpage对象都有一个template_name属性可以自定义模板的名字，但是默认情况下Flatpages应用会在所有的模板里查找一个叫做flatpages/default.html的模板。这就是说你要在项目的TEMPLATE_DIRS设置里列出的某个位置里创建一个“flatpages”模板目录，或者如果你使用app_directories模板加载器的话，就要在INSTALLED_APPS设置下列出的某个应用程序里的“templates”文件夹中建立它。不管是哪种方式，现在先把它建好。

和你希望的一样，你的模板是由一个叫做flatpage的对象传递进来的。例如：

```
<h1>{{ flatpage.title }}</h1>
<p>{{ flatpage.content }}</p>
```

现在在你刚刚创建的目录创建一个最简单的页面模板并保存为default.html：

```
<html>
    <head>
        <title>My Dummy Site: {{ flatpage.title }}</title>
    </head>
    <body>
        <h1>{{ flatpage.title }}</h1>
        <p>{{ flatpage.content }}</p>
    </body>
</html>
```

测试一下

好了，现在试着去载入你的flatpage页面。例如，如果服务器是运行在你自己的机器上，并且你通过admin创建的Flatpage对象的URL值为 /about/ 的话，你可以在浏览器里输入 http://localhost:8000/about/ 。它应该会用default.html模板显示title和content变量的值，如图8.3所示。

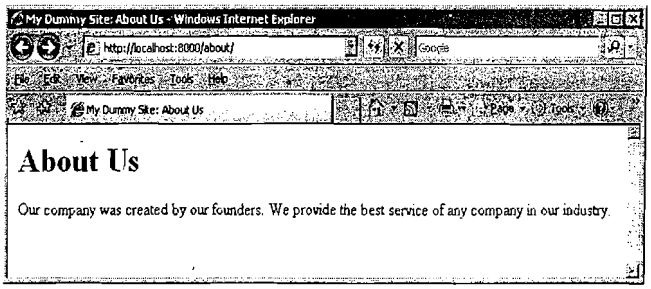


图8.3 一个“关于”Flatpage的例子

到此你已经体验过了Flatpages的用法以及它适用的方位。下面开始是本章的重头戏——一个更强壮的自定义CMS应用示例。

8.3 超越Flatpages：一个简单的自定义CMS

虽然Flatpages很精致，但是前面也说了，它们也有诸多的限制。要超越它的难易程度则取决于你对站点的需求。我们来看看用Django来构建一个比Flatpages更好的自定义CMS所需要的过程。特别地，我们希望这个方案具有以下特性。

- 允许用户输入非HTML格式的文本，稍后会自动转换成HTML。
- 依据可读的文字来创建页面URL而不是面向数据库主键。
- 提供基本的工作流——能将一个员工用户和文章联系起来，同时也允许把文章标属为某个产品上的某个部分。
- 维护创建和修改的时间。
- 为文章提供分类功能，并且能按照这些类别来浏览文章。
- 为所有页面提供一个简单的搜索功能。

这些功能都可以通过Django来完成，其中多数所需的特性你在之前都已经看过了。学习构建Django应用在一定程度上也就是学习如何尽可能有效地把这些特性组合起来以达成所要的结果。

首先，我们要再设置一个Django项目（如果不记得怎么创建项目，数据库等内容的话，请复习一下第2章，“Django速成：构建一个Blog”）。我们给项目起名为cmsproject，它只包含了一个应用cms。

先从模型（model）开始。

创建模型

下面是我们这个小型CMS的核心模型定义。注意这里引用了两个其他的模型（User和Category），稍后我们就会看到它们的定义以及添加一些必要的import语句。

```
class Story(models.Model):
    """A hunk of content for our site, generally corresponding to a page"""

    STATUS_CHOICES = (
        (1, "Needs Edit"),
        (2, "Needs Approval"),
        (3, "Published"),
        (4, "Archived"),
    )

    title = models.CharField(max_length=100)
    slug = models.SlugField()
    category = models.ForeignKey(Category)
    markdown_content = models.TextField()
    html_content = models.TextField(editable=False)
    owner = models.ForeignKey(User)
    status = models.IntegerField(choices=STATUS_CHOICES, default=1)
    created = models.DateTimeField(default=datetime.datetime.now)
    modified = models.DateTimeField(default=datetime.datetime.now)

    class Meta:
        ordering = ['modified']
        verbose_name_plural = "stories"

    @permalink
    def get_absolute_url(self):
        return ("cms-story", (), {'slug': self.slug})

class StoryAdmin(admin.ModelAdmin):
        list_display = ('title', 'owner', 'status', 'created', 'modified')
        search_fields = ('title', 'content')
        list_filter = ('status', 'owner', 'created', 'modified')
        prepopulated_fields = {'slug': ('title',)}

admin.site.register(Story, StoryAdmin)
```

模型类定义里的第一块代码定义了一个包含四个阶段的工作流。当然，你可以给自己的流程添加更多的步骤。

虽然使用Django映射来完成变量选择（就像STATUS_CHOICES展示的一样）有很多方便的地方，但是在这里它在数据库里最后还是归结为整数。由于稍后要再重新定义“1”代表什么意思就不是那么容易了，所以最好先停下来想一想，确认你的列表是不是合理。当你要根据这个变量对模型实例进行排序的时候就更是如此了，而我们这里正有理由要这么做。

我们还要在公共视图里用这些值来决定访客能看到什么，即，他们可以看到“Published”

和“Archived”的文章，但是看不到“Need Edit”或是“Needs Approval”。这就是由商业，项目，和/或者应用程序需求所决定的逻辑。

如果不想把这样的一个列表硬编码在代码里，你可以转去使用ManyToManyField，这样就可以和其他数据一样在admin里编辑了。

定义完STATUS_CHOICES之后就轮到变量的定义。

- title：我们要在浏览器的标题栏和渲染页面上显示的标题。
- slug：页面在其URL里唯一的名字。这比一个无意义的整数主键要好多了。
- category：文章的类别。这是一个指向稍后定义的另一个模型的外键。
- markdown_content：Markdown格式的页面正文（下面会专门介绍Markdown）。
- html_content：HTML格式的页面文本。我们会在编辑的时候自动渲染它，所以在显示页面的时候就不会有标记翻译的开支了。为了避免混淆，不可以直接编辑这个变量（所以不会在Django的admin应用里的编辑表单上显示出来）。
- owner：拥有这个内容的admin用户（或者，从Django的角度来说，这是一个指向User对象的外键引用）。
- status：在编辑工作流中的状态。
- created：创建的时间，自动设置为当前时间（利用Python的datetime模块）。
- modified：修改的时间，初始化为当前时间。我们需要一些特殊的步骤才能保证在文章被修改时会自动更新。这个时间戳会显示在文章详细页面上。

我们对这个模型所做的一个修饰（纯粹是为了admin的用户），就是在Meta嵌套类里指定了一个verbose_name_plural属性。这能让我们的模型不会在admin应用里显示“Storys”这样错误的名字。最后，还有一个生成永久链接的get_absolute_url方法，在第7章“Photo Gallery”里我们已经见过它了。

Imports

除了django.db.models（以及稍后要解释的一个关联的permalink装饰器函数），我们还需要导入的有datetime模块（用于created和modified变量）和来自Django的contrib.auth应用的User模型。此外还有用来把我们的模型向admin应用注册的Django admin模块。

```
import datetime
from django.db import models
from django.db.models import permalink
from django.contrib.auth.models import User
from django.contrib import admin
```

和Flatpages应用一样，当你开始构建高级的Django应用时就会发现User模型缺少了很多功能。例如，它组成用户名字的方式可能和你的要求有冲突。但是，User特别地好用，算是一个中等完备的解决方案，而且在很多现实世界的应用里也是非常有用的。

完成模型

User对象是直接从Django贡献的“auth”应用里拿来的，那么Category呢？它倒是我们自己的，以下是其模型定义，它应该出现在models.py里Story模型定义的上面。

```
class Category(models.Model):
    """A content category"""
    label = models.CharField(blank=True, max_length=50)
    slug = models.SlugField()

    class Meta:
        verbose_name_plural = "categories"

    def __unicode__(self):
        return self.label

class CategoryAdmin(admin.ModelAdmin):
    prepopulated_fields = {'slug': ('label',)}

admin.site.register(Category, CategoryAdmin)
```

Category模型相当简单，甚至有点琐碎。在Django的应用程序里，你经常会看到这类简单的模型（有时候简单到只定义了一个变量）。不过要是把它变成Story模型里的一个“category”变量的话，那会让事情变得困难起来（给category改名），甚至失去某些功能（比如，给category添加描述属性等）。Django既然提供了方便的途径来创建适合的关系型模型，那么最好还是照做吧。

对于Story，我们还设置了一个verbose_name_plural属性，这样admin的用户就不会笑我们连拼写都不会了。

控制文章的显示

我们的数据库同时包含了已发布的（之前STATUS_CHOICES里的3和4）和还未发布的（状态1和2）文章。我们需要一种便捷的方式只把前者显示在站点的公共页面上，而在admin里则显示全部。因为这里涉及的是商业逻辑而非表现风格，所以应该在我们的模型里实现它。

虽然我们也可以通过模板里的{% if ... %}标签来达成，但是这个方案最终会变得异常繁琐和不断重复。（如果你不信的话，我们建议你不妨试试看——说不定在你完成前就受不了这种缺陷了！）根据本书作者的共同经验，永远不要把商业逻辑放到模板里面去，否则时间一长必定搅得一团乱。

我们把这项功能通过自定义Manager加到Story模型里去。关于该技巧更多的细节，请参见第4章，“定义和使用模型”里的“自定义Manager”一节。在models.py文件的import语句下面加上如下代码：

```
VIEWABLE_STATUS = [3, 4]
```

```
class ViewableManager(models.Manager):
    def get_query_set(self):
        default_queryset = super(ViewableManager, self).get_query_set()
        return default_queryset.filter(status__in=VIEWABLE_STATUS)
```

首先我们把VIEWABLE_STATUS定义一个整数列表，它的值正对应了可以让公众看到的文章的状态。这是一个模块级的属性，即将来添加其他方法的时候也能够访问到它。

接着，我们在模型里实例化一个manager对象。在models.py文件的底部（跟在变量和Meta嵌套类定义的后面），加上下面两行，注意这里代码要进行适当的对齐，让它属于Story类：

```
admin_objects = models.Manager()
objects = ViewableManager()
```

就像在第四部分里提到的那样，因为admin_objects是先定义的manager，所以它就是我们模型的默认manager，admin也会用它——这样就能保证所有阶段的文章都可以被员工修改。这里的名字除了提醒我们它的作用外并无特殊之处。

然后我们用常见的objects为名创建一个自定义manager的实例。因为我们在URLconf和视图里用的都是这个名字，因此所有公共的页面都自动收到由自定义ViewableManager提供的特殊的过滤的文章queryset。

使用Markdown

作为模型的最后一个部分，我们重写了内置函数save来应用一个轻型的标记语言，叫做Markdown，用户在admin里用它来输入文本。Markdown和Wiki风格的语法颇为相似，它提供了一种创建Web内容更简单的方式。编写Markdown比原始HTML要愉悦得多了，任何写过纯文本email或是编辑过Wiki页面的人都不会对它感到陌生。

你还可以选用Textile，ReStructuredText等其他轻型的标记语言。这里我们主要展示的是如何通过重写模型的save方法来“神奇自动”地把Markdown转换成HTML，这样就不需要为每个页面请求执行一次翻译了——我们在前面描述markdown_content和html_content变量的时候已经提到过这一点了。

> **为什么不用WYSIWYG？**
>
> 既然基于Web的内容管理系统一般都是针对非技术背景用户的，有人心里或许会犯嘀咕，我们这里用Markdown是不是有点太nerd了。确实，若是在Django admin里集成WYSIWYG（所见即所得）的HTML编辑器的话，用户可能会更容易上手。
>
> 不过除了要多花一点精力实现以外，这个方法的另一个缺点在于WYSIWYG还是不能把Web浏览器变成Microsoft Word，而且还容易出现浏览器不兼容的问题。不过说归说，这类工具还是能够增加像CMS这样的工具的吸引力和对非技术用户的粘度。要得到WYSIWYG插件的推介和其他的建议，请访问withdjango.com。

如果要在Python里使用Markdown，你得先去下载Python-Markdown模块，因为它不是标准

库的一部分。你可以在http://www.freewisdom.org/projects/python-markdown/ 找到它。安装完成后，用下面的语句把markdown函数从markdown模块里导进来：

```
from markdown import markdown
```

以上的import看着有点绕圈，不过其实这在Python里是很常见的一种手法，即模块和从模块里要导入的属性名有着相同的名字。

不懂Markdown并不会妨碍你理解这个应用程序，不过这里还是为初学者准备了几个例子。你要是喜欢的话可以在Python的解释器里自己试验一番。在这个例子里，我们定义了一个tidy_markdown帮助函数，它能去掉Markdown插入到输出里的换行符号（\n），从而使打印结果更清晰明了。（当我们用Markdown输出更多的HTML时，这些换行就能防止输出超级长的一行。）

```
>>> from markdown import markdown
>>> def tidy_markdown(text):
...     return markdown(text).replace('\n', '')
>>>
>>> tidy_markdown("Hello")
'<p>Hello</p>'
>>> tidy_markdown("# Heading Level One")
'<h1>Heading Level One</h1>'
>>> tidy_markdown("Click here to buy my book (<http://withdjango.com/)">)
'<p><a href="http://withdjango.com/">Click here to buy my book</a></p>'
>>> tidy_markdown("""
... An alternate H1 style
... ====================
... > A blockquote
... * Bulleted item one
... * Bulleted item two
... """)
'<h1>An alternate H1 style</h1><blockquote><p>A blockquote</p></blockquote><ul>
 <li>     Bulleted item one </li>
 <li>     Bulleted item two </li></ul>'
```

这里输入的是以Markdown语法写成的纯文本，而函数的输出则是合法的HTML。

回到我们的Django应用：要把Markdown内容自动转换为HTML以便保存，我们还要在模型代码里多加上一点代码。这是一个只有三行的函数，把它放在admin_objects赋值的那一行之上就行了（注意要保持和模型类里其他代码的对齐）。

```
def save(self):
    self.html_content = markdown(self.markdown_content)
    self.modified = datetime.datetime.now()
    super(Story, self).save()
```

当我们的代码（或者其他任何使用我们模型的应用，比如Django admin）试图把一个对象保存到数据库里时，模型的save方法会先被调用，将用户输入的Markdown内容翻译成HTML。（如果你要复习一下super调用的语法，可以参考第1章，“实战Django Python”。）

纯数据库主义者可能不太喜欢这种内容可以轻易地从另一个字段里计算得来的字段。如果这一转换没什么计算成本的话，我们就不需要保存这个渲染出来的HTML。这也是每个不同的项目经常要做出取舍的地方。这里我们假设计算能力是有限的资源，比如一个异常繁忙但是内容却不一定很多的站点。而对于更关注数据库大小的站点（比如一个有着几千乃至几百万条记录的社区论坛），每次页面访问才去计算可能是比较好的选择。

因为保存HTML的模型变量被标志为editable=False，所以它不会显示在admin的界面里。这让用户的交互更加清晰，并且消除了用户改动并保存而覆盖渲染出来的HTML的可能性，因为这种覆盖可能带来破坏。所有的修改都是由Markdown源码完成，转换成HTML后保存到html_content变量里，无需任何关注。另外在保存的时候，我们还一并把modified变量更新为当前的时间戳。

要了解更多Markdown及其语法，请参见官方网站http://daringfireball.net/ projects/markdown/。另外Python-Markdown自己也有一些非常有用的第三方扩展。事实上，本书正是在WrappedTables "wtables"（参见http://brian-jaress.livejournal.com/5978.html）的帮助下用Markdown编写完成的！如果你有兴趣的话，可以参阅另一个Python的Markdown项目，见http://code.google.com/p/python-markdown2/ 。

urls.py里的URL模式

重写完save函数后，模型这一块就算是完成了。在讨论视图和模板之前，我们先来看一下URL。这里是项目层里的urls.py。

```
urlpatterns = patterns('',
    url(r'^admin/(.*)', admin.site.root),
    url(r'^cms/', include('cmsproject.cms.urls')),
)
```

admin这行还是一样，另一行则把所有的URL匹配到CMS应用的"cms/"上去。如果你还需要其他前缀，比如"stories"或是"pages"，当然也是可以在此处指定的。想了解其他灵活的根URL设置方式，请参考第7章。

下面这个文件是上面代码里include调用的应用层urls.py：

```
from django.conf.urls.defaults import *
from cms.models import Story

info_dict = { 'queryset': Story.objects.all(), 'template_object_name': 'story' }

urlpatterns = patterns('django.views.generic.list_detail',
    url(r'^(?P<slug>[-\w]+)/$', 'object_detail', info_dict, name="cms-story"),
    url(r'^$', 'object_list', info_dict, name="cms-home"),
)

urlpatterns += patterns('cmsproject.cms.views',
    url(r'^category/(?P<slug>[-\w]+)/$', 'category', name="cms-category"),
    url(r'^search/$', 'search', name="cms-search"),
)
```

我们的URL按照顺序提供了单篇文章的显示，整个文章的列表，分类文章的列表，配了某个搜索请求的列表。

由于这里我们再次利用了Django的通用视图，所以基本上涵盖了应用程序全部的功根据不同的视图前缀，我们把四个URL模式分到两个patterns对象里去，不过我们本也导入和使用这些视图函数。

注意

这里我们用的是字符串因为我们想要在填充自定义视图前就使用admin和通用视以在这种情况下试图导入还未定义的函数是不行的。不过这种在URL里使用字符串数对象通常是相对比较随意的，只要适合你的需要就好。

就像在之前章节里看过的那样，通用视图提供了很多可选的参数来控制它们的这里我们只用其中一个，template_object_name，它让我们可以在模板里用story来引对象而不是默认的名字object。

Admin视图

现在CMS应用的admin站点应该能够正常工作了。（别忘了运行一下manage.py syncdb数据库帮你把表都建好。）访问这个网站，登录后你应该可以看到如图8.4所示的admin图8.5是点击Add后所显示的Add Story页面。

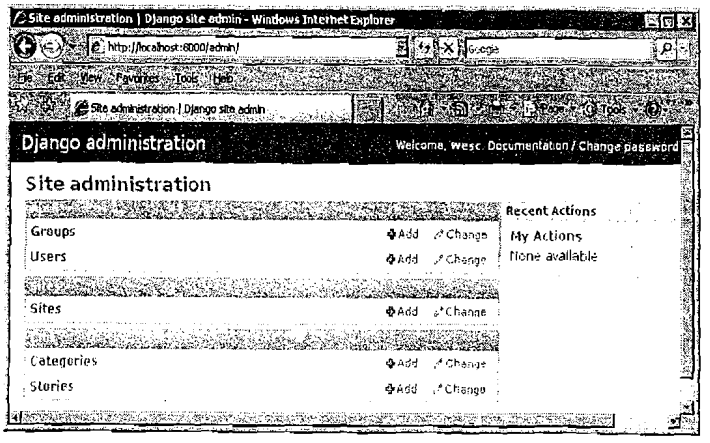


图8.4　admin页面

你还可以选择创建一个类别。如果你点击Add Story页面上的“+”号，就可以看到所示的一个小的弹出窗口。

例如，在Label变量里输入“Site News”，即可同时在Slug变量看到一个为Web应用字符串了（如图8.7）。

接着我们就可以继续完成我们的文章了。在该例子中，我们把状态设置为Public图8.8）。

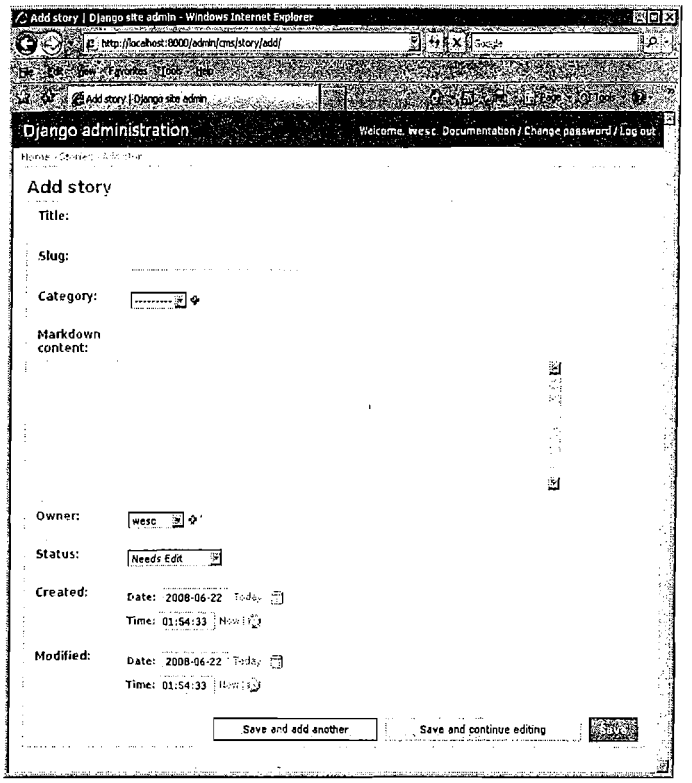


图8.5 在admin里添加一篇文章

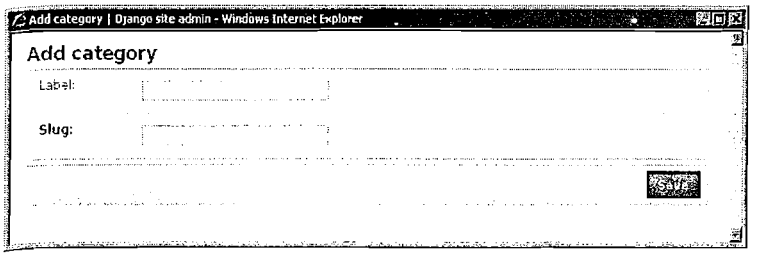


图8.6 在添加文章的同时加入一个类别

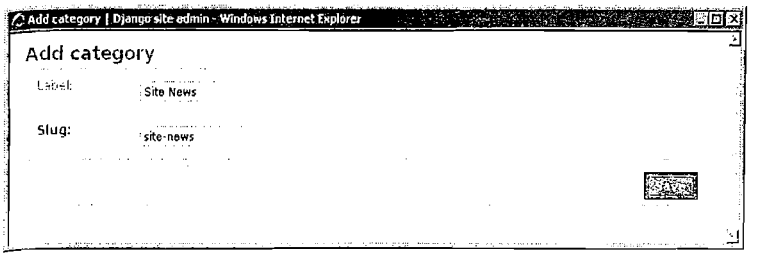


图8.7 加入“Site News”类别

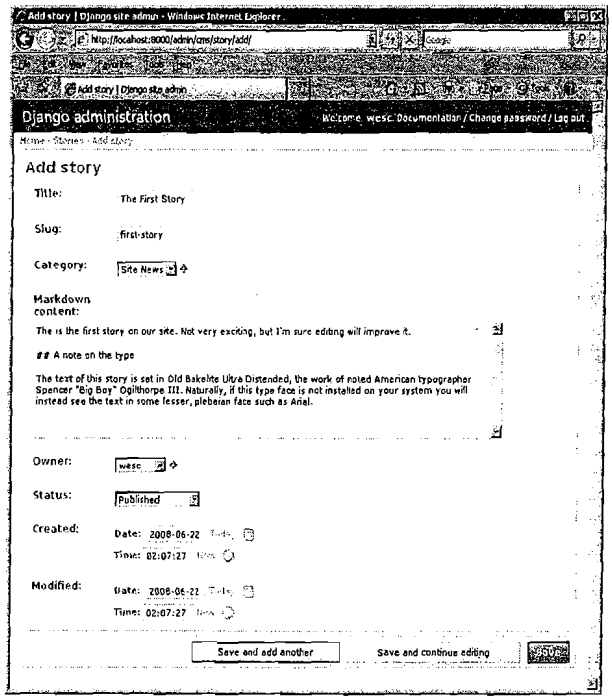


图8.8　完成我们的第一篇文章

按下保存按钮，你会被重定向到CMS的Story页面（如图8.9），此时应该可以看到刚刚添加的文章了。

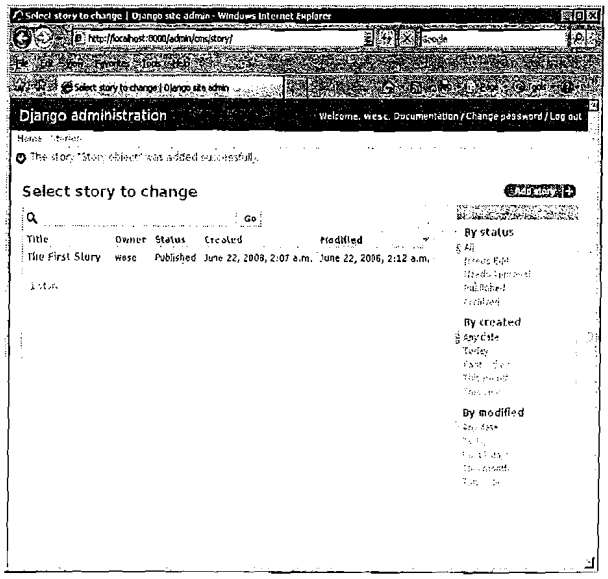


图8.9　在admin里浏览文章的列表；注意右边可选的过滤器

试着再随便添加编辑一些文章，确保Published或Archived状态的文章至少各有一篇，这样

我们才能在站点里有东西可看。

使用通用视图显示内容

在前面的URLconf里可以看到，我们前台显示的部分基本上用的都是通用视图。不过在分类列表显示里我们还需要一点自定义视图。以下是该应用程序的riews.py文件的起始部分。

```
from django.shortcuts import render_to_response, get_object_or_404
from django.db.models import Q
from cms.models import Story, Category

def category(request, slug):
    """Given a category slug, display all items in a category."""
    category = get_object_or_404(Category, slug=slug)
    story_list = Story.objects.filter(category=category)
    heading = "Category: %s" % category.label
    return render_to_response("cms/story_list.html", locals())
```

以上是一个相当简单的视图函数，不过却完成了现有的通用视图处理不了的功能，这也是为什么我们要自己写的原因。接着我们将继续介绍模板，并在稍后再回到第二个自定义视图上来，让它提供一个搜索界面。

模板布局

几乎所有的Django项目里都有一个base.html，所有其他的模板都由它扩展而来。这里我们只要扩展两个即可：story_detail.html和story_list.html。在cms文件夹里创建这三个文件，并在settings.py文件里为它们设置TEMPLATE_DIRS。

首先是最简单的base模板：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
    <head>
        <title>{% block title %}{% endblock %}</title>
        <style type="text/css" media="screen">
            body { margin: 15px; font-family: Arial; }
            h1, h2 { background: #aaa; padding: 1% 2%; margin: 0; }
            a { text-decoration: none; color: #444; }
            .small { font-size: 75%; color: #777; }
            #header { font-weight: bold; background: #ccc; padding: 1% 2%; }
            #story-body { background: #ccc; padding: 2%; }
            #story-list { background: #ccc; padding: 1% 1% 1% 4%; }
            #story-list li { margin: .5em 0; }
        </style>
    </head>
    <body>
        <div id="header">
            <form action="{% url cms-search %}" method="get">
```

```
                <a href="{% url cms-home %}">Home</a> &bull;
                <label for="q">Search:</label> <input type="text" name="q">
            </form>
        </div>
        {% block content %}
        {% endblock %}
    </body>
</html>
```

稍后我们再解释模板里Django相关的部分。现在，我们需要一个能显示一篇单独文章（story_detail.html）的模板，就像刚刚所说，它是从base模板扩展而来。

```
{% extends "cms/base.html" %}
{% block title %}{{ story.title }}{% endblock %}
{% block content %}
    <h1>{{ story.title }}</h1>
    <h2><a href="{% url cms-category story.category.slug %}">{{ story.category
}}</a></h2>
    <div id="story-body">
        {{ story.html_content|safe }}
    <p class="small">Updated {{ story.modified }}</p>
    </div>
{% endblock %}
```

这在可用的模板里大概算是最简单的一个了——它只需要一个story模板变量。只要传给它一个有title和html_content属性的对象，它就能够正常工作。

该模板中非常重要的一点就是应用在html_content变量上的safe过滤器。默认情况下，Django会自动转义模板里所有的HTML 以防止用户输入的恶意信息（Web应用里日益严重的安全性问题）。因为我们的Markdown源码都是由可信赖的用户输入的，所以我们有理由相信其内容是“安全的”，因此允许浏览器直接处理HTML，而不用把转义为等。

好几个需要显示多篇文章的视图都要用到我们的列表模板story_list.html，比如分类列表，搜索结果以及主页。

```
{% extends "cms/base.html" %}
{% block content %}
    {% if heading %}
        <h1>{{ heading }}</h1>
    {% endif %}
    <ul id="story-list">
    {% for story in story_list %}
        <li><a href="{{ story.get_absolute_url }}">{{ story.title }}</a></li>
    {% endfor %}
    </ul>
{% endblock %}
```

以上就比之前的细节模板稍微复杂了一点。它循环取出story_list里的每一项，为它们创建元素并且把标题作为链接的文本，而链接则是指向文章的细节页面。

显示文章

因为我们用到了slug变量，所以显示在开发服务器上的URL就是http://localhost:8000/cms/first-story/。重新启动服务器，并在浏览器里输入以上URL，结果如图8.10所示。

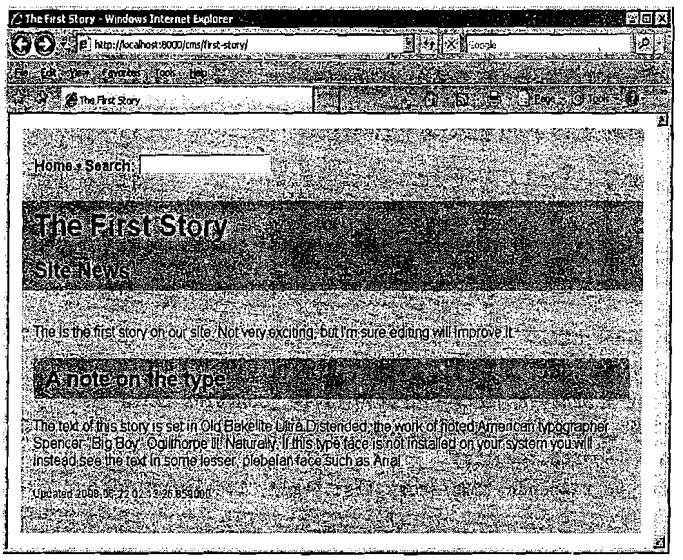


图8.10 我们的第一篇文章的“细节页面”

接着，我们访问站点的主页来测试object_list视图。它的URL为http://localhost:8000/cms/。如图8.11所示。

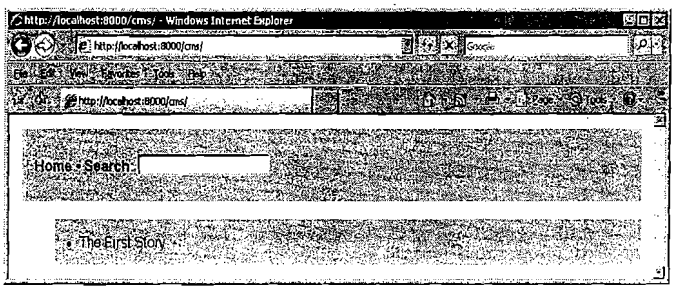


图8.11 列出所有文章的主页

文章的标题即是链接，它们是由之前创建的get_absolute_url方法生成的。

注意到页面上方的搜索框了么？下面我们就要让它工作起来。

添加搜索功能

搜索文章内容的功能是必须要有的。对于一个公共站点来说，你只需要加上一个Google Site-Search框就行了（http://www.google.com/coop/cse/），但是如果能对搜索过程以及结果的表现进行更多控制的话那就更好了。

现在我们来给网站加上一个简单的搜索功能，只需要以下几个步骤就能完成。

- 给base.html模板加上一个包含搜索字段的HTML表单，这样每个页面上都能看到它
- 添加一个接受该表单输入，并且查找匹配文章的视图函数
- 至于用来显示结果的story_list.html模板，我们已经创建好了

前面的base.html模板，它在头里包含了一个搜索框。要它起作用的话，我们就需要在表单被提交上来的时候专门有一个视图来处理它。

这项任务没法用通用视图完成，所以我们需要另一个简单的自定义视图函数。在views.py文件里的category方法定义的下面加入以下代码：

```
def search(request):
    """
    Return a list of stories that match the provided search term
    in either the title or the main content.
    """
    if 'q' in request.GET:
        term = request.GET['q']
        story_list = Story.objects.filter(Q(title__contains=term) |
➥(markdown_content__contains=term))
        heading = "Search results"
    return render_to_response("cms/story_list.html", locals())
```

虽然这是一个自定义视图，但是它并不需要一个专用的模板。我们可以直接把story_list.html模板拿来重用，只要我们提供它所需的参数就行了——即在上下文变量story_list里一个由Story模型对象组成的QuerySet。搜索算法也非常简单，只要能在标题或是Markdown正文里能找到通过表单提交上来的文字，那么那个Story就是匹配的。

我们来多添加几个“stories”。在该例子中，我们加入一个“About Us”页面（就像在Flatpages里那样）并把它标记为Archived。然后再添加一个Contact Us页面，不过需将它保留在Needs Edit状态。我们的admin页面现在应该会显示所有三篇文章以及它们的状态，如图8.12所示。

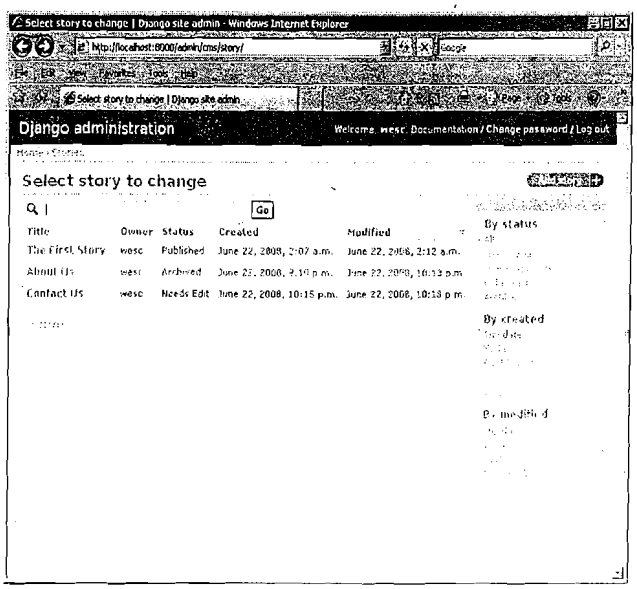


图8.12　列出全部文章的admin页面

但是在主页上，我们应该只显示能公开访问的页面及其链接（Published或Archived，这两个都是由VIEWABLE_STATUS控制的）。当访问主页时，如图8.13所示，注意此时Contact Us页面没有显示！

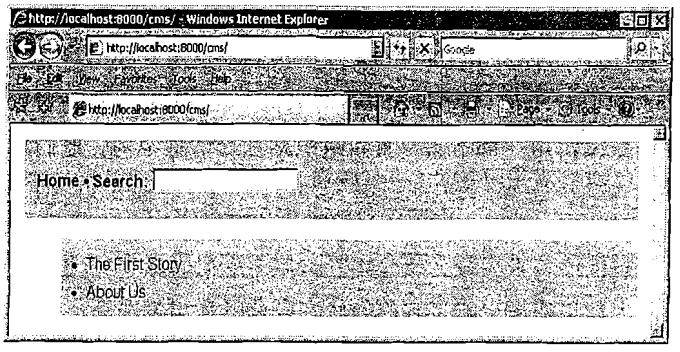


图8.13 显示可公开访问的文章的主页

现在来试一试搜索功能。在搜索框里输入“typographer”，我们可以看到只有第一篇文章是匹配的，如图8.14。

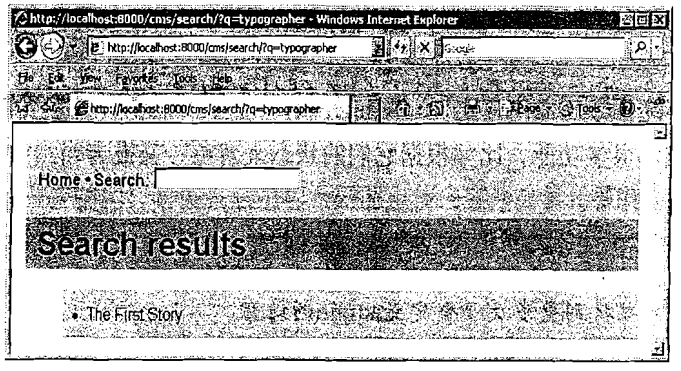


图8.14 搜索结果页面（列出的还是只有可公开访问的文章）

以上就是我们实现的CMS里全部的核心功能。下面要讨论的是应用程序里的最后一个部分：管理用户和权限，以及有商业逻辑控制的工作流。

管理用户

我们的系统提供了一个拥有权的概念，每篇文章都和一个特定的Django用户对象相关联。技术上一个用户完全可以修改或删除不属于他的内容，事实上甚至不会阻碍他们修改拥有权字段。这个字段并没有建立起任何之前没有的对象访问控制。

> **注意**
>
> 在不久的将来，Django可能会实现一个更细致的“对象权限”系统，在本书编写的时候，这项新的admin特性还在开发之中。想了解更多信息，请访问withdjango.com。

尽管如此，这种不正规的松散的拥有权管理方式在相互信任的组织内部还是颇为有用的。这就像在一个办公室环境里，你不会怀疑别人偷了你漂亮的红色订书机或者把你的文件扔进碎纸机里去一样。这里实现拥有权的方便之处在于我们利用了Django内建的用户模型。我们不需要其他任何额外的模型代码。所以我们可以用Django admin来管理用户。

作为admin的超级用户，你可以用admin来控制谁能编辑用户和用户组，谁又有权限来访问你的Story模型。你还可以让用户只能编辑Story对象但不能编辑Category对象。这种限制还是有必要的，因为绝大多数编辑用户都不需要重新组织站点的信息架构，他们只要能添加或者更新现有的内容就好了。

支持工作流

以下这个简单的内容工作流影响了我们的状态变量以及相关选择：

1. 一个外部的作者或员工提交了页面的内容。这份内容处于草稿形式，并且需要被编辑。
2. 编辑初稿完成后，在发布前还需要终审。
3. 一旦文章被标为“Published”后，随即便出现在公共网站上。
4. 如果文章过期了，它会被标为“Archived”。这就是说，它会出现在站点搜索的结果里，但是不会出现在首页上“Recent articles”的列表中。

这一章的例子没有涉及任何自定义admin的内容。如果有的话，自定义视图里可能会大量使用这个变量来完成很多功能，比如根据它们的状态来着色，又或者当用户登录admin时向他们显示可以执行的操作等。

> **注意**
> 你可以在第11章里找到更多关于自定义admin的内容。

注意我们的models.py使用了Django admin里的list_filter特性来方便地选择处于四个状态下的任何文章。例如，编辑用户可以用它来选择所有处在Needs Edit状态下的文章，而一名专门负责删除旧资料的实习生可以专注于Archived状态下的文章。

8.4 改进建议

在本章开始的地方我们就提过，CMS架构的种类非常多。你在本章里构建的应用示例根据所需的特性不同，最终可以演变成各种样子。以下是一些建议。

分页。在只有十几二十篇文章的时候，我们的列表页看着还算可以。但是当这个数量达到成百上千，把它们全部显示在一页里就有点吓人了，而且这还可能会拖累网站的性能。同样，如果一个搜索返回了几百个结果，用户也不太可能希望一次性全部看到它们。好在Django内置了一些对分页的支持，即django.core.paginator模块。想了解更多信息，请查阅Django官方文档。

更强大的搜索。我们的搜索功能还算能用，但是它和我们熟悉的Web搜索引擎比起来还差

的很远。例如，多个单词的短语除非特别指定，否则最好被当作是一组独立的搜索词。这里的实现还可以做的更复杂，不过要是想要对大量记录进行全文搜索的话，你应该去看看Sphinx这种可以集成到Django里的搜索引擎。想了解更多信息，请访问withdjango.com。

状态改变提示。我们已经在一个自定义的save方法里处理了Markdown的渲染。我们还可以进一步扩展它来改进我们的工作流系统，比如监测文章状态的变化，并给负责那篇文章的人发送提示email。实现这个功能的关键在于将我们的状态变量通过ForeignKey替换成一个全功能的Status模型，除了在STATUS_CHOICES里的数字值和名称之外，再加一个指向User模型的ForeignKey变量status_owner。save方法就可以对status的值和要保存的值进行比较，如果发现不同，那就调用Django的send_mail函数去通知相应的用户。

动态生成导航。我们的应用程序除了默认列出全部文章之外，没有提供别的站点的导航方式。对一个真实的网站来说，这样是不够的。一种可行的办法就是在Story模型里加上一些和导航有关的变量。而更灵活的方案则是提供一个额外的Navigation模型，它可以只包含三个变量：在整个导航序列里的位置，要显示给用户的标签，和一个导航对象应该指向的文章的ForeignKey。

用户评论。我们的CMS可以发布内容，但是最终用户却无法贡献内容。所以一个比较自然的想法就是允许用户评论某篇文章。好消息是Django内置了一个出色的评论系统，无论是注册用户还是匿名用户都可以使用它。坏消息是在编写本书的时候，这个系统正在大规模地重写，所以我们没办法把它包括进来。不过如果你对这个功能有兴趣的话，可以去看看官方文档，一旦这个特性完成，我们马上就能看到它了。

静态文件。很多营销和公共关系组织都希望能上传内容来分发给现有的和潜在的客户，包括提供演讲，报告和技术白皮书等，它们可以是各种格式的文件，比如PDF，Word文档，Excel表格，ZIP压缩文件等。

8.5 总结

这一章很长，但是你应该看到了如何利用多个Django核心组件和contrib应用来构建简单的Flatpages站点以及更复杂的CMS。

到此，我们希望你已经熟悉了Django应用构建的方式：通过命令行工具创建一个项目和应用，考虑好模型的定义（包括怎么使用admin），定义URL，使用通用和自定义视图，以及创建一个模板层次。

这一部分还有两个更重要的应用示例：一个使用了Ajax技术来创建一个Liveblog，另一个则是基于Django的Pastebin。

第9章 Liveblog

本书的重要内容就是利用Django开发Web应用。正如你已经看到的，Django提供了丰富的内置功能，所以你无需彷徨失措就能做到很多事情。但是和所有的工具一样，Django也不是万能的，它缺乏的最明显的一个Web功能就是没有集成Ajax，即异步JavaScript和XML（Asynchronous JavaScript And XML）。

不过这也意味着Django并没有把你绑在某个Ajax库上（市面上这种库不计其数），而是把选择权留给了你。

在这一章里，我们要展示一个相对简单的Ajax应用，即“liveblog”。一个liveblog是一个列出了一系列简短、带时间戳的条目的网页，它能进行自我更新而无需用户干预。要是你对近年来Apple的新闻比较关心的话，你应该已经在各种Mac新闻和流言的站点（比如macrumorslive.com）上见过这类应用了。同样的概念也可以较小范围地用在普通，静态的blog上，虽然它也用同样的方式提供即时信息，不过通常没有动态更新。

以下的应用示例会向你介绍在Django Web应用里集成Ajax所需的一切知识，同时又不会过多涉及复杂的C/S交互或是动画。我们还会在不牵涉具体工具的情况下，指出Django在哪些方面可以和Ajax携手合作。

> **注意**
>
> 和其他应用示例一样，这里我们也用Apache来简化静态文件的处理（对于这个例子来说就是JavaScript）。

9.1 究竟什么是Ajax

当在Web开发里提到“Ajax”的时候，通常指的是两种不同但又经常纠结在一起的行为。

- 无需用户重新载入或者访问他处就能让网页获取额外的信息——想想GMail是怎么在浏览器不重载或是重绘整个页面的情况下显示email，收件箱以及表单的。
- 高级“动态的”用户接口——想想Google Maps的地图滚动和放大缩小，或者各种支持“widget”拖放接口特性的个人主页站点。

从实现层面上来说，可以把Ajax里“额外信息”的部分想象成迷你的请求，是浏览器和服务器里一个无需重新载入整个页面的普通HTTP对话。稍后我们会介绍这里面的细节，现在先注意这些对话的响应部分通常都是XHTML或XML（即Ajax里的“X”），又或者是轻型的数据格式，比如JSON。

Ajax的UI部分其实就是漂亮的客户端JavaScript和DOM操作，这些技术是由于近来越发强

的浏览器和客户端电脑才变得现实起来。若是考虑到样式正确的Web标记元素显示的可能性，JavaScript本身是一门完整的编程语言这一事实，网页现在其实已经和传统的GUI动画技术很相像了。

为什么Ajax很有用

对程序员来讲，一个网页发出迷你请求的能力从几个方面来看都很有用。它能在大流量的情况下节约带宽，因为客户浏览器只要求了特定部分的数据而非整个页面，另外它创造了一种响应更好的用户体验，因为浏览器窗口不用不停地重绘。这让Web应用感觉起来更接近桌面应用程序。

虽然有时候它们被批评成“华而不实”，但是只要使用得当，高级的动画、拖放功能以及其他“Web 2.0”的特性都可以大大增强用户体验。迷你请求带来的减少页面重载的次数和良好集成的动画和特效，都进一步模糊了Web和传统GUI之间的界线。

设计应用程序

在深入代码之前，我们要给出一个简单的规格，应用程序要有什么特性，以及决定用什么工具（特别是用哪个Ajax库）来构建它。首先我们先写下一些需求，定义出应用程序要达到的功能。

- 应用程序只含有一个网页。不需要其他花哨的东西——我们只是要建立一个一次更新一个事件的liveblog。
- 它只跟踪一条连续的信息流。尽量保持简单。
- 这个“流”由带时间戳的文本段落组成。所以我们的模型只需要两个变量。
- 这条“流”按时间逆序显示，即最新的放在最前面。所以最新的信息总是显示在页面顶端。
- 初始页面载入显示的是流的当前状态。访问一个不断变化的liveblog的用户无需Ajax就可以看到到那一刻为止的所有事件。
- 页面每隔一分钟就会向服务器提出新的请求。这时就需要Ajax出场了。
- 事件是通过Django admin添加的。不过若是有需要，为它创建一个自定义的表单也不难——为了得到更出色的响应，你甚至可以在后台也使用Ajax来提交。

选择一个Ajax库

在编写本书的时候市面上可用的Ajax JavaScript库有好几种，它们各有千秋，专注点也不尽相同。有些提供了大量的UI widget，其他的则固守阵地，尽量把JavaScript变得更加容易使用。它们还采用了不同的方式来更新JavaScript，提供了自有的语法来操作网页的HTML结构。

很多库都分成好几个组件以供下载，分成专注于JavaScript语言更新的“核心”部分，负责迷你请求的“网络”部分，提供UI widget的“widget”部分，当然还有包含一切的“完整版”。所以除了选择工具集，你还要知道那个工具的作用并下载正确的版本。

组织媒体文件，甚至都不一定非要把它放在项目文件夹里。我们只是出于开发拥有大量JavaScript，CSS文件，和图片的大型网站的习惯，在这里专门准备了一个独立的js子目录。在该例子中，我们没有使用外部的CSS或者图片，但是如果有的话，就还可以放置img和css文件夹。

把媒体放在Django项目文件夹内部可以让它在服务器以及源码控制里的管理简单许多。把media文件夹符号链接到Apache的document root下（要把Apache配置成允许AllowSymlinks）可以确保里面的媒体文件都能正常工作。

我们的liveupdate应用里，模型，URL，和视图各有一个文件。根据给定的需求，我们的URL和模型都非常简单，而且现在还没有任何非通用视图。下面是应用程序层上的URLconf文件liveupdate/urls.py（它应该被include在项目层的urls.py里），它只列出了Update对象：

```
from django.conf.urls.defaults import *
from liveproject.liveupdate.models import Update

urlpatterns = patterns('django.views.generic',
    url(r'^$', 'list_detail.object_list', {
        'queryset': Update.objects.all()
    }),
)
```

现在我们来看看models.py文件，它定义了Update模型类（包括默认的排序方法）并为它设置使用admin：

```
from django.db import models
from django.contrib import admin


class Update(models.Model):
    timestamp = models.DateTimeField(auto_now_add=True)
    text = models.TextField()

    class Meta:
        ordering = ['-id']

    def __unicode__(self):
        return "[%s] %s" % (
            self.timestamp.strftime("%Y-%m-%d %H:%M:%S"),
            self.text
        )

admin.site.register(Update)
```

最后是我们初始的模板（templates/update_list.html），它是用户首次进入站点时看到的“静态的”视图，显示了我们更新列表当时的状态：

```
<html>
    <head>
        <title>Live Update</title>
```

```
        <style type="text/css">
            body {
                margin: 30px;
                font-family: sans-serif;
                background: #fff;
            }
            h1 { background: #ccf; padding: 20px; }
            div.update { width: 100%; padding: 5px; }
                div.even { background: #ddd; }
            div.timestamp { float: left; font-weight: bold; }
            div.text { float: left; padding-left: 10px; }
            div.clear { clear: both; height: 1px; }
        </style>

    </head>
    <body>
        <h1>Welcome to the Live Update!</h1>
        <p>This site will automatically refresh itself every minute with new
        content  - please <b>do not</b> reload the page!</p>

        {% if object_list %}
            <div id="update-holder">
                {% for object in object_list %}
                <div class="update {% cycle even,odd %}"id="{{ object.id }}">
                    <div class="timestamp">
                        {{ object.timestamp|date:"Y-m-d H:i:s" }}
                    </div>
                    <div class="text">
                        {{ object.text|linebreaksbr }}
                    </div>
                    <div class="clear"></div>
                </div>
                {% endfor %}
            </div>
        {% else %}
            <p>No updates yet  - please check back later!</p>
        {% endif %}
    </body>
</html>
```

从逻辑的角度来讲，该模板相当的直白，也没有包含任何JavaScript或JavaScript includes。在下一节里，我们会用jQuery来给模板加入动态的部分。

不过在这之前，我们先测试一下，在给它添加核心功能前先感受一下。激活Admin（如果你不记得怎么做了的话，请参考之前的章节），运行manage.py syncdb，启动Apache，然后访问admin。

你应该可以看到和往常一样的包含了我们Update模型类的admin控件，点击Add然后像图9.1那样填写一些内容。注意因为我们在timestamp变量里指定了auto_now_add，所以我们只需要输入文本就好了——这些都是为了更快的更新liveblog。

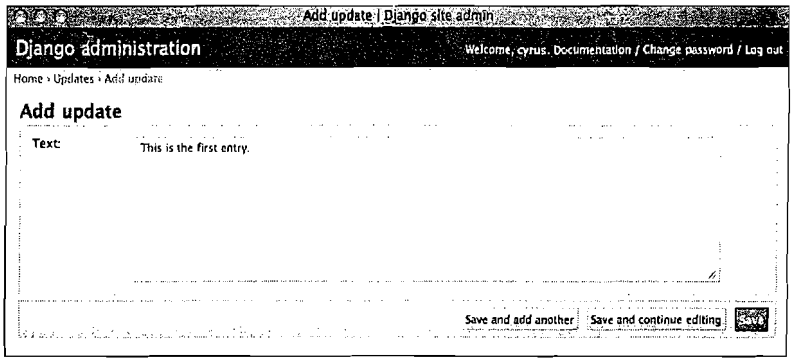


图9.1 添加一个新的Update

添加完之后，admin就会显示新加入的Update，如图9.2所示。

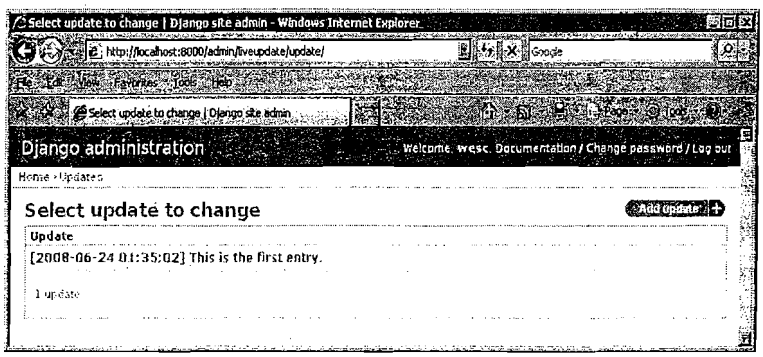


图9.2 登录包含Liveupdate应用的Admin页面

点击前端的根URL，你还可以看到一个“静态”版本的liveblog，如图9.3所示。现在应用程序的基础已经准备好了，我们要开始应用Ajax了。

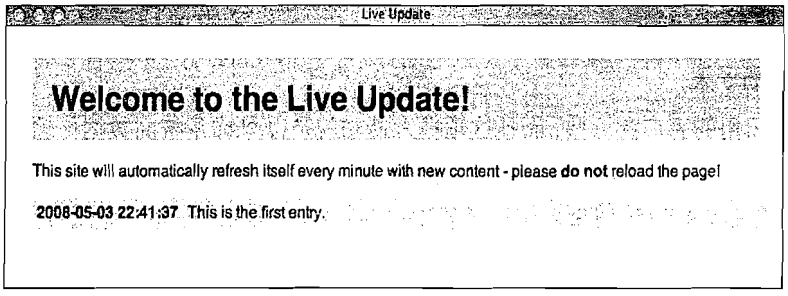


图9.3 只有一条消息的liveblog

9.4 加入Ajax

这一节占据了本章剩下的绝大部分篇幅，但是别气馁！在使用Ajax的时候，你需要很多背景知识才能完全理解到底是怎么回事，和往常一样，我们希望能给你一个学习的机会而不是直

接告诉你怎么复制粘贴。

首先是简单的浏览一下使用Ajax需要什么，以及它用什么格式来传递信息，然后是安装和测试拥有Ajax功能的JavaScript库，最后展示实际化腐朽为神奇的代码，包括服务器端和浏览器端的代码。

基本概念

实际上来说，网站“实现Ajax”主要包含了以下三个主要任务。

- 导入类库：因为我们主要是用第三方库来实现功能，所以必须在使用它之前导入模板。
- 定义客户端回调函数：我们用导入的库来编写一个向服务器发出迷你请求的函数，并用返回的结果更新网页。
- 定义服务器端逻辑：最后，服务器需要知道如何响应迷你请求，所以我们必须定义一个知道怎么做的Django视图。处理Ajax的视图就是普通的视图（即它也是一个接受HTTP请求然后返回HTTP响应的函数），有时也会有一到两个额外的包头（见本章稍后的“创建视图函数”一节）。

第一步（导入类库）通常就是一个JavaScript包含，而那两个函数是简单还是复杂则要取决于你逻辑的需要。最典型的，客户端会做比较多的工作，而服务器端经常就是JavaScript和数据库之间的一座桥梁而已，不过根据你和你用户的特定需要这些都是可以改变的。另外，如果你有兴趣使用Ajax库里UI部分的话，不管它是不是和迷你请求一起并发执行的，那些代码都应该是在模板这一层上。

Ajax里的“X”（XML v.s. JSON）

客户端和服务器端函数之间的对话理论上可以包含任何JavaScript代码能处理的格式（当然也得是HTTP可以传输的格式）。但是因为所需的结果一般都是要把HTML传输或是添加到网页上去，所以绝大部分的Ajax对话（就像前面提到的）都是采用XML（XHTML是它的一种变体）或是一种文本数据格式JSON（JavaScript Object Notation），即一种简单的可以被转换成JavaScript变量的文本。

按理说XML用的更广泛一些，而且这项技术最初就是以号称系统间数据传输语言而闻名的。而且由于XML和（X）HTML之间亲密的关系，它也非常适合做这项工作，因为JavaScript和其他Web开发工具都是被设计用来操作这种层次化的数据结构的。此外，你还可以在服务器端就格式化好HTML（例如，用Django的模板和渲染引擎），这样客户端的代码就只需要一个函数把它放到合适的地方就行了。

JSON是最近才兴起的一种格式，它有简明可读的语法，而且比XML消耗的带宽更少。另外还有一个优点就是它的语法和Python的数据结构（字符串，字典和列表）惊人的相似。更多关于JSON语法的信息可以在http://json.org找到。

以下是JSON数据结构的一个简单的例子。

```
{"first": "Bob", "last": "Smith", "favorite_numbers": [3,7,15]}
```

即使你一点也不懂JavaScript，运用你的Python背景也能理解这是一个拥有两个字符串值和一个整数列表值的字典。这样的字符串会被转换成JavaScript里的数据结构，随后可以被客户端代码使用。

> **JSON变Python**
>
> 很多情况下，JSON都可以被直接转换成Python数据结构，这在客户端JavaScript给服务器发送数据时很有用。但是它和Python还是有一点不兼容的地方，比如JSON的true和false布尔值和Python的就不一样（Python里是首字大写的），以及JSON的null和Python的None。在这种情况下，使用一个Python/JSON解析器就很有必要了。要知道更多JSON和Python互操作的信息，可以访问上面列出的JSON网站以及这两篇文章http://deron.meranda.us/python/comparing_json_modules/ 和http://blog.hill-street.net/?p=7。

在这个应用程序里我们采用的是JSON，但是XML依然是一个很流行的选择。无论是书籍还是网络，这两种格式都有大量的范例，帮助文档和教程。

安装JavaScript库

因为我们已经决定在这个例子使用jQuery了，所以我们先要从http://jquery.com上把它下载下来。jQuery就是一个单独的类库，而不像其他库那样分成多个部分，但是它还是提供了多种下载方式——最小化的，打包的和压缩的。三个包的功能都是一样的，只不过文件大小不同，而且需要客户端花费不等的CPU时间来解压。

我们采用的是jQuery当前最小化的版本1.2.6，不过任何1.2.x版都能在我们的代码里工作。(1.2.x之前的版本应该也能工作，但是它们缺少getJSON函数，所以需要一条额外的语句来转换JSON字符串。)

jQuery需要在我们的模板里应用，所以它要和自定义的JavaScript一起放在liveproject/media/js 目录下。在Win32机器上，只需要从浏览器里把文件下载到那个文件夹即可。而在Unix系的主机上，比如Mac OS X和Linux，我们可以用wget或curl命令行工具直接下载文件，比如（用浏览器的复制功能把下载URL抓下来）：

```
user@example:/opt/code/liveproject $ cd media/js/
user@example:/opt/code/liveproject/media/js $ wget
http://jqueryjs.googlecode.com/files/jquery-1.2.6.min.js
 -2008-05-01 21:52:15 -  http://jqueryjs.googlecode.com/files/jquery-1.2.6.min.js
Resolving jqueryjs.googlecode.com... 64.233.187.82
Connecting to jqueryjs.googlecode.com|64.233.187.82|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 54075 (53K) [text/x-c]
Saving to: `jquery-1.2.6.min.js'

100%[=====================================================>] 54,075
227K/s   in 0.2s
```

```
2008-05-01 21:52:16 (227 KB/s) - `jquery-1.2.6.min.js' saved [54075/54075]
```

把类库保存到media目录下以后，我们需要在<head>标签里加入下面这行代码，这样就能把它包含到模板里来了（注意 /media/custom 是Apache docroot里一个指向我们liveproject/media目录的符号链接）。

```
<script type="text/javascript" language="javascript"
    src="/media/custom/js/jquery-1.2.6.min.js"></script>
```

至此使用jQuery前的准备工作就完成了，我们先设置一个简单的测试，然后才是实现真正的功能。

设置和测试jQuery

Ajax库通常都提供了一种简单的方式来访问它的代码，当前浏览器的文档，或当前的DOM对象。jQuery在它几乎所有的功能里都使用了一种相对独特的语法。变量名$被绑定到一个特殊的可调用对象上，它可以被当作函数一样调用（例如$(argument)），又可被用来保存特殊的方法（比如$.get(argument)）。

例如，$(document)返回一个通常可以和普通的JavaScript document变量相比较的对象，但是jQuery还给它加了点料。其中一个额外的方法就是ready，当页面载入时就会用它去执行一个JavaScript函数（同时避免了JavaScript内置onLoad函数的问题）。

举例来说（并为我们最终的函数做准备），在模板中的<head>标签里，包含jQuery的代码后面加上下面的代码：

```
<script type="text/javascript" language="javascript">
    $(document).ready(function() {
        alert("Hello world!");
    })
</script>
```

JavaScript和Python一样，函数是“first-class”的或普通的对象。在高级JavaScript开发中会大量修改对象以及传递函数。在这里，$(document).ready接受一个函数并在页面准备好的时候执行它，而且我们当场匿名地定义了这个函数——这个Python里lambda的手法很像，不过这里它能够把函数写成多行。如果一切正常，而且模板能包含jQuery的话，我们的JS代码就会在你刷新页面的时候弹出一个对话框显示“Hello World!”。这只是最简单的例子，后面还有更加精彩的实现效果。

在模板里嵌入JavaScript

为了让事情变得有趣一点，我们把以上代码改一下，在载入页面的时候给我们的列表动态地加上一个 （来表示一个Update对象，虽然这有点硬编码了）。当然，这和我们稍后即时更新代码所做的事情是一样的。

```
<script type="text/javascript" language="javascript">
    $(document).ready(function() {
        $("#update-holder").prepend('<div class="update">\
```

```
                <div class="timestamp">2008-05-03 22:41:40</div>\
                <div class="text">Testing!</div>\
                <div class="clear"></div>\
            </div>');
    })
</script>
```

这里我们使用了jQuery的选择器语法，它允许你向$()查询函数传递一个类似CSS的字符串，然后函数返回的结果就是一个代表了所有匹配对象的Query。在这里，我们查找的是之前模板里定义的 标签，所以使用了CSS里用来选择对象ID的#字符。

选中之后，我们调用prepend方法把一个对象或HTML字符串加到选中内容之前——所以这里它在列表的开头又加上了一个<div class="update"。就单行的JavaScript来说，还不算太糟。我们来看看载入页面后的结果，如图9.4。

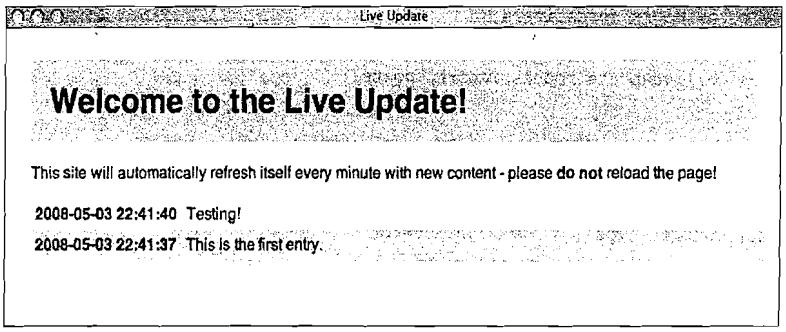


图9.4 测试我们即时更新的JavaScript

到此测试就完成了——jQuery已经正确安装。现在可以回到服务器端来创建我们的Django视图了。我们将实现一个小巧的数据服务API，然后就可以为最后一步进入jQuery的请求机制了。

创建视图函数

要实现这个需求特性（让我们的JavaScript能请求所有比当前的显示更新的内容）最方便快捷的方法就是设置一个普通的URL。如果我们要在请求里发送多个信息，那么我们就可能需要基于POST的API，但是这里我们可以采用简单一点的方案。

假设我们Ajax视图的URL是 /updates-after/<id>/，这里<id>是请求JavaScript能见到的最新的ID。我们的视图只需要根据这个ID号去做一个简单的查询，然后返回所有比它新的Update对象就可以了。返回的格式是一个JSON编码版本的模型对象，这样客户端的JS就可以轻松地对其解析并将其包裹到HTML里去。

> **时间戳 v.s. ID**
>
> 这里实际上有两种排序Update对象的方法：根据ID和根据时间戳字符串。根据ID比较简单（不需要在查询时把时间戳字符串解析成Python的datetime对象），也更加细致（根据Update更新到数据库里的频率——你可能会碰到很多相同时间戳的情况！）。但是，使用ID

这个方法假设的是ID永远都是自动增长的，就像Django里的自动ID一样——可是现实世界里并不一定总是如此。

当然，技术上知道怎么做是一回事，而知道怎么正确地把需求翻译成实实在在可行的方案又是另一回事了，通常那都要比看起来棘手的多。

在应用程序的URLconf文件liveupdate/urls.py里加入以下这行代码。

```
liveupdate/urls.py.

url(r'^updates-after/(?P<id>\d+)/$',
    'liveproject.liveupdate.views.updates_after'),
```

以下是liveupdate/views.py里响应的视图函数。

```
from django.http import HttpResponse
from django.core import serializers

from liveproject.liveupdate.models import Update


def updates_after(request, id):
    response = HttpResponse()
    response['Content-Type'] = "text/javascript"
    response.write(serializers.serialize("json",
        Update.objects.filter(pk__gt=id)))
    return response
```

利用Django内置的序列化库我们省掉了不少麻烦，它能把模型对象翻译成任意一种文本格式，包括XML，JSON以及YAML。serializers.serialize函数接受一个QuerySet，根据主键（pk）选择对象——在这里，我们只需要ID比传入的参数id更大的那些对象。

然后函数按照我们选择的格式返回字符串结果（这里是JSON），我们再把它写到HttpResponse里去。最后，要让JavaScript能正确解析和使用响应的正文，还需要在响应里设置Content-Type头。

使用可读的序列化文本格式的好处就是我们可以轻松的调试它们。图9.5显示的是在浏览器里手动访问URL http://localhost:8000/updates-after/0/ 时的结果，这时数据库里只有一个测试用的Update。

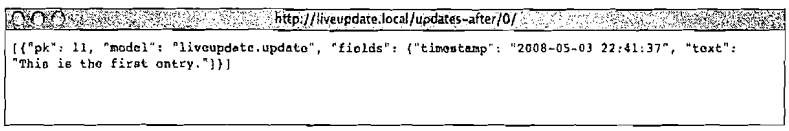


图9.5　在浏览器里测试JSON视图

我们马上就要大功告成了——最后一步是重中之重，即编写和这个API视图交互的JavaScript，并用它的响应来更新页面。

通过JavaScript调用视图函数

JavaScript提供了一个内置的计时函数用来在一定的时间间隔里重复执行任意代码，它就是setInterval，它接受一个要执行的函数名称字符串和一个毫秒的时间间隔，比如：

```
setInterval("update()", 60000);
```

这就会每隔60000毫秒或60秒执行一次update()函数。加上另一个好用的jQuery方法getJSON（它会向一个URL发出迷你请求，并将结果解析为JSON），我们就最终完成了Ajax。这里就是最终的代码：

```
<script type="text/javascript" language="javascript">
    function update() {
        update_holder = $("#update-holder");
        most_recent = update_holder.find("div:first");
        $.getJSON("/updates-after/" + most_recent.attr('id') + "/",
            function(data) {
                cycle_class = most_recent.hasClass("odd")
                    ? "even" : "odd";
                jQuery.each(data, function() {
                    update_holder.prepend('<div id="' + this.pk
                        + '" class="update " ' + cycle_class
                        + '"><div class="timestamp">'
                        + this.fields.timestamp
                        + '</div><div class="text">'
                        + this.fields.text
                        + '</div><div class="clear"></div></div>'
                    );
                    cycle_class = (cycle_class == "odd")
                        ? "even" : "odd";
                });
            }
        );

    }
    $(document).ready(function() {
        setInterval("update()", 60000);
    })
</script>
```

update内部的逻辑应该相当清晰明了，以下是一些小结：

1. 我们通过jQuery的各种选择方法来获取容器对象（update_holder）和最近更新的条目（most_recent）。

2. most_recent的HTML ID属性（方便起见我们就用服务器端数据库的ID来填充）被用来构建URL，它也是getJSON的第一个参数。

3. 第二个参数通常是匿名函数，它包含了下面的几点。

4. 函数的第一行初始化了“even/odd”CSS类变量。

5. 然后用jQuery的each函数迭代视图里的JSON数据，得到一个序列化的Update对象列表。

6. 把那些要用来构建新的HTML块的Update对象加到 容器里。

7. 最后，在循环结束的地方循环这个CSS类来改变行的颜色。

定义完update之后，在ready里实际执行的代码就是前面提到的setInterval而已。在发布额外的blog条目后，你就可以在保存后一分钟内看到它们被自动加载到网页上。虽然我们没办法实际演示这些代码（动画GIF或视频不太适合以书本形式显示），图9.6是经过若干次更新后的站点的截图。

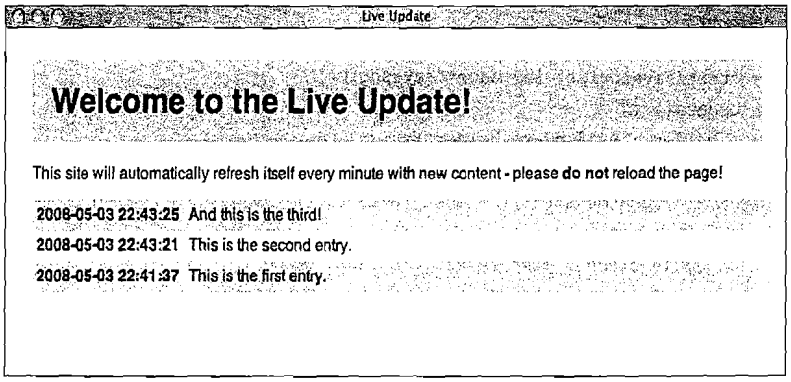


图9.6　包含多条消息的liveblog最终的形态

9.5 总结

虽然我们使用了jQuery强大的选择语法和它的getJSON函数来提供客户端的功能，但对在这里设置的服务器端来说，这两项技术都不是必需的。这不仅仅是因为绝大多数其他的Ajax数据库也都有类似的工具，而且实际上不依赖任何特殊的类库而直接使用我们自己的API视图也是相当简单的。

这里关键的地方在于所有的组件都是通过HTTP来交流的（我们的视图接受的常用的GET HTTP请求，换成POST当然也是可以的），而且（经由HTTP发送的）返回值也是一个公开的格式，JSON。就和Django内部的组件尽量保持灵活和模块化一样，配合Ajax使用Django也是依赖于两者都具备开放和良好定义的特性。

这一章更多的关注于Ajax里迷你请求的部分，但是还有很多阅读资料介绍了JavaScript在UI前端能做到的各种惊人效果。我们推荐你可以读一下附录D，“发现，评估，使用Django应用”，记住——虽然在应用程序里加入很多花哨的附属功能很诱人，但是用户还是会希望能够恰到好处。

第10章 Pastebin

对很多程序员来说，Django最大的吸引力之一就是可以用Python来开发Web应用。而同时，它提供的通用视图又能让你不用写什么代码就可以创建Web应用。（通用视图的内容在第5章，“URL，HTTP机制，和视图”里。）

Django 的通用视图是强大的快速开发和建模工具。不过不要被它的名字给骗了，通用视图并不仅仅是用完即拆的脚手架而已。Django新手难免会以为它是那种能显示数据的默认模板，这都是可以理解的，但事实上并非如此。记住，Django的视图就是一段接受一个HttpRequest请求并返回一个 HttpResponse响应的Python代码。怎么把数据对象传递给视图，以及怎么用模板渲染这个响应，完全是取决于你自己的设计。

在这一节里，我们将介绍一个依赖于通用视图的简单的pastebin应用。这里是它的一些特性

- 一个含有一项必填项（内容）和两项可选项（张贴者的名字和一个标题）的表单
- 一个显示最近条目的可点击列表
- 每个条目的详细视图
- 一个允许我们（站点的拥有者）编辑或删除现有条目的管理员界面
- 语法着色
- 定期清除过期的条目

我们几乎不用写太多Python代码就能完成所有这些功能。我们只要创建一个描述数据及其属性的模型文件和要显示它的模板，除此之外的其他部分就都由Django代劳了。

> **注意**
>
> 这个应用示例的代码和Django社区的pastebin的第一版dpaste.com很相似。虽然今天的dpaste.com不再是一个纯通用视图的应用了，但是它简单和实用的特性依然保留在本章的代码里。

在我们继续前还要注意一点：这个例子的精髓在于展示框架能为我们做多少事情。虽然有人会说这也太懒了，不过不是你写的代码就代表了也不用你去调试。所以在该例子中，当遇到选择是多写一点代码来获得多一些自定义行为还是让Django帮我们处理的时候，我们还是交给Django去做了。

10.1 定义模型

以下是pastebin应用程序完整的models.py。它定义了一个有五个变量的数据结构，一些

Meta选项，和一组通用视图和模板要用到的方法。它还把模型注册给admin并设置了一些admin里和列表相关的选项。

```
import datetime
from django.db import models
from django.db.models import permalink
from django.contrib import admin

class Paste(models.Model):
    """A single pastebin item"""

    SYNTAX_CHOICES = (
        (0, "Plain"),
        (1, "Python"),
        (2, "HTML"),
        (3, "SQL"),
        (4, "Javascript"),
        (5, "CSS"),
        )

    content = models.TextField()
    title = models.CharField(blank=True, max_length=30)
    syntax = models.IntegerField(max_length=30, choices=SYNTAX_CHOICES, default=0)
    poster = models.CharField(blank=True, max_length=30)
    timestamp = models.DateTimeField(default=datetime.datetime.now, blank=True)

    class Meta:
        ordering = ('-timestamp',)
    def __unicode__(self):
        return "%s (%s)" % (self.title or "#%s" % self.id,
        ➥self.get_syntax_display())

    @permalink
    def get_absolute_url(self):
        return ('django.views.generic.list_detail.object_detail',
            None, {'object_id': self.id})

class PasteAdmin(admin.ModelAdmin):
    list_display = ('__unicode__', 'title', 'poster', 'syntax', 'timestamp')
    list_filter = ('timestamp', 'syntax')

admin.site.register(Paste, PasteAdmin)
```

这里大多数内容都很面熟了，不同的是它们组成了这个pastebin应用里大部分自定义的Python代码。除了在应用程序urls.py里设置的一些简单规则，大部分的工作都由框架负责完成了。

像这样基于通用视图的应用程序真正展示了Django的DRY（Don't Repeat Yourself，不要重复自己）理念的威力。在我们这个将要构建的例子中，上面核心模型里定义的五个变量

(content, title, syntax, poster, 和timestamp) 分别用于:

- 创建数据库里的数据表,即manage.py syncdb命令
- admin应用,它为我们的数据生成了一个编辑界面
- object_detail通用视图,它从模型里获取实例,然后把它们发送给模板系统去显示
- create_update通用视图,它生成并处理了新添加的paste条目所提交的表单

另外还有好几个地方都用到了模型里的方法。admin应用会在它引用对象名字的时候(比如,在确认删除的消息里)用到__unicode__对象,而 get_absolute_url方法则负责生成“View on site”的链接。我们的模板还在所有需要显示条目名字的地方都隐式调用了__unicode__,它还使用了get_absolute_url来为最近条目的列表生成链接。

10.2 创建模板

现在我们来创建一些基本的模板把内容渲染出来显示给用户。首先我们需要一个基础模板,这种手法在之前的第7章里已经见过了。把以下文件保存在pastebin/ templates下。

```
<html>
<head>
<title>{% block title %}{% endblock %}</title>
<style type="text/css">
    body { margin: 30px; font-family: sans-serif; background: #fff; }
    h1 { background: #ccf; padding: 20px; }
    pre { padding: 20px; background: #ddd; }
</style>
</head>
<body>
    <p><a href="/paste/add/">Add one</a> &bull; <a href="/paste/">List all</a></p>
    {% block content %}{% endblock %}
</body>
</html>
```

有了基础模板之后,我们来创建一个允许用户向我们的应用程序粘贴代码的表单。把下列代码保存为pastebin/templates/pastebin/paste_form.html。(下面会解释为什么这里有一个看似多余的路径名。)

```
{% extends "base.html" %}
{% block title %}Add{% endblock %}
{% block content %}
<h1>Paste something</h1>
<form action="" method="POST">
Title: {{ form.title }}<br>
Poster: {{ form.poster }}<br>
Syntax: {{ form.syntax }}<br>
{{ form.content }}<br>
<input type="submit" name="submit" value="Paste" id="submit">
</form>
{% endblock %}
```

接着是显示列表的模板。它显示了所有最近张贴的条目，让用户可以从中点击选择。把它保存为pastebin/templates/pastebin/paste_list.html。

```
{% extends "base.html" %}
{% block title %}Recently Pasted{% endblock %}
{% block content %}
<h1>Recently Pasted</h1>
<ul>
    {% for object in object_list %}
    <li><a href="{{ object.get_absolute_url }}">{{ object }}</a></li>
    {% endfor %}
</ul>
{% endblock %}
```

最后是详细页面的模板。人们绝大多数时间都会停留在该页面上。把它保存为pastebin/templates/pastebin/paste_detail.html。

```
{% extends "base.html" %}
{% block title %}{{ object }}{% endblock %}
{% block content %}
<h1>{{ object }}</h1>
<p>Syntax: {{ object.get_syntax_display }}<br>
Date: {{ object.timestamp|date:"r" }}</p>
<code><pre>{{ object.content }}</pre></code>
{% endblock %}
```

10.3 设计URL

因为我们这个应用程序的结构非常清晰，所以为之创建URL也相当地简单。惟一巧妙的就是利用到通用视图的部分。我们需要设计三个URL模式——一个列出所有的条目，一个显示单独的条目，还有一个负责添加新条目。

```
from django.conf.urls.defaults import *
from django.views.generic.list_detail import object_list, object_detail
from django.views.generic.create_update import create_object
from pastebin.models import Paste

display_info = {'queryset': Paste.objects.all()}
create_info = {'model': Paste}

urlpatterns = patterns('',
    url(r'^$', object_list, dict(display_info, allow_empty=True)),
    url(r'^(?P<object_id>\d+)/$', object_detail, display_info),
    url(r'^add/$', create_object, create_info),
)
```

以上就是我们应用程序的核心了。我们从django.views.generic导入了三个函数对象：

- django.views.generic.list_detail.object_list

- `django.views.generic.list_detail.object_detail`
- `django.views.generic.create_update.create_object`

除了所有Django视图（不管是不是通用的）都接受的第一个参数HttpRequest外，它们收到了一个额外的字典值，我们在之前的URLconf里定义了两个不同的字典。它们的名字（display_info和create_info）在这里虽然是随便取的（虽然_info是这些字典惯用的后缀），但是它们的内容却是为我们的通用视图专门定制的。list_detail视图接受一个包含所有合法对象的queryset。而create_update视图接受的则是模型类（不是实例）。在object_list里，我们给字典加上allow_empty=True来告知视图即使数据库里没有对象我们也要显示这个页面。

在这个字典里我们还可以包含很多其他可能的值。因为这是自定义通用视图的主要手段，所以它们的数目还是很大的。要了解这些视图完整的可选项，你可以参考Django官方文档。现在我们先尽量保持简单。

> **注意**
>
> 在你大量使用这些_info字典的时候，有一个值得注意的特殊规则。不是每个请求都会去对URLconf里的代码求值。就是说，如果没什么特别的事情的话，我们在这里设置的Paste.objects.all查询集有可能会在新对象被用户或网站管理员添加、编辑，或删除后过期。好在Django意识到了这一点，并且能明确指示不要缓存这些关键的queryset数据。

10.4 试运行一下

虽然还没有写下很多代码，不过我们已经有了一个可以工作的应用程序了。现在来跑一下看看。启动应用程序后，我们可以看到如图10.1所示的画面——一个空的pastebin。

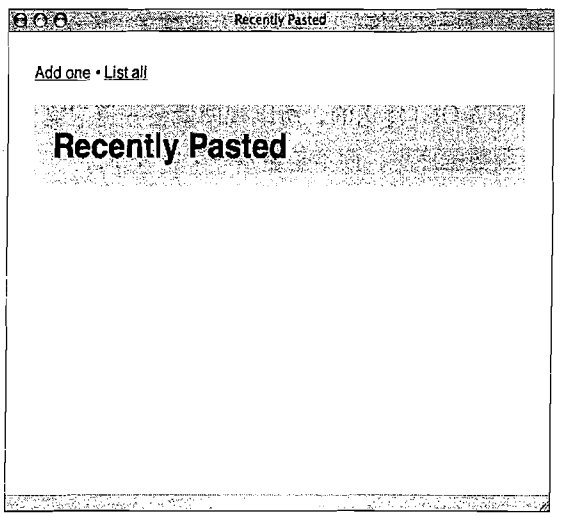


图10.1 空白的Pastebin

我们来想一下获得这个页面时Django都为我们做了哪些事情。首先它解析了我们的URL请

求，确定要调用哪一个视图，传递一个从我们的模型继承而来的（空的）查询集，找到响应的模板文件，用适当的上下文渲染之，最后把结果HTML返回给浏览器。

现在加上一点内容。点击Add One链接，我们应该能看到一个空的表单。这个表单是Django视图和我们的模板通力合作的结果。通用视图create_update审视了我们的模型，生成HTML表单元素，并把他们传递给模板里的{{ form }}模板变量。我们的模板把这些元素展开后将表单显示给用户。注意<form>标签和提交按钮都是由我们负责的。

这个表单看起来应该如图10.2所示。

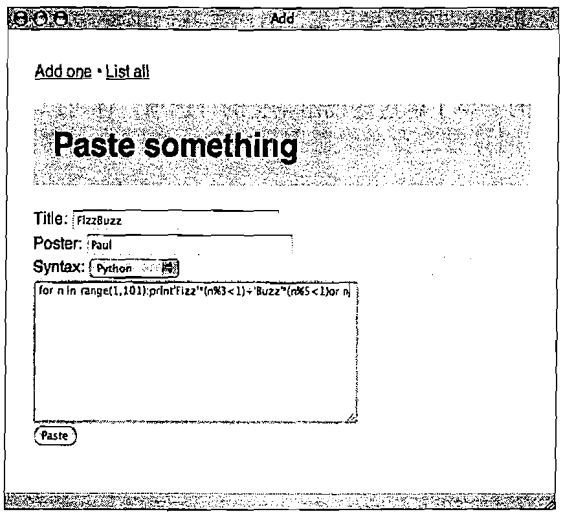


图10.2　Add One表单

我们友好的pastebin不会逼迫用户做太多的事情。事实上这里只有Code字段是必须的。Pastebin只有方便才会好用，一长串必填的字段那就太麻烦了。

填入内容后点击Paste按钮，它就会去再次调用Django的create_update通用视图。因为这里数据是经由HTTP POST而不是GET传送的，所以视图就会知道它需要去处理用户的输入并且（如果可能的话）把它保存到数据库里，而不是显示一个空表单。

这里create_update没有展示的一个地方就是验证。要是用户忽略了必须的变量怎么办？很简单，这个表单会再次显示。在我们极端简化的模板里没有包含查找或显示验证错误消息的代码，但实际上Django确实在{{ form.errors }}模板变量里传递了它们。更强壮的做法是查找这些错误并把它们用友好的方式反馈给用户。

假设用户正确填写完表单并点击Paste按钮，Django就会（再次通过create_update通用视图）处理表单输入并保存到数据库里。然后，它会重定向到新建对象的get_absolute_url并且用pastebin_detail.html模板把我们提交的内容渲染出来，如图10.3所示。

这通常是用户提交后最希望看到的结果——即那个提交的视图和一个可以发送给别人的URL。

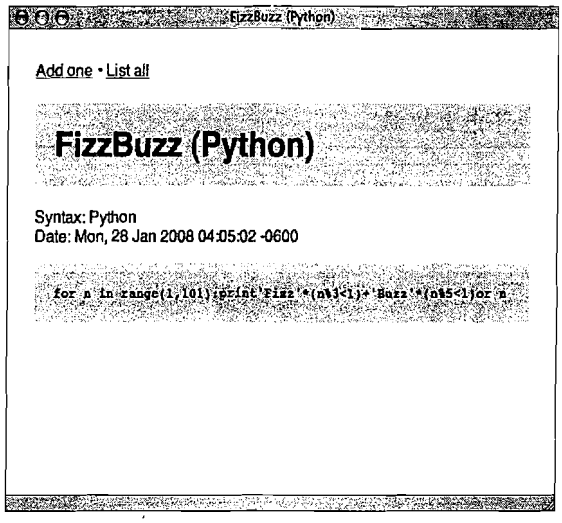


图10.3 新张贴的条目

如果用户要浏览pastebin里其他的条目。List All链接会把他导向正确的位置。Django的object_list通用视图和paste_list.html模板会显示一个简单的列表，如图10.4。

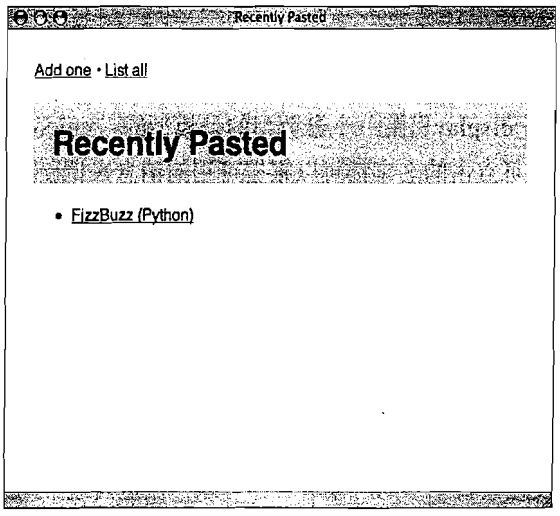


图10.4 提交条目的列表

这个变量漂亮地展示了object_list通用视图的能力。URLconf里的display_info字典把代表所有Paste对象的queryset传递给object_list视图。视图再将对象传给模板，而模板再通过一个简单的{% for object in object_list %}循环生成可点击的列表。

注意

在实际中，列出所有提交的条目对这类网站来说未必是一个很好的选择。Pastebin用户通常只关心他们自己提交的东西。有一个关于Pastebin的笑话就是，“为什么老有人往我的

pastebin里贴一些乱七八糟的东西？”回答是，“制造垃圾信息的人不会‘关心’他们给无辜的用户发送什么内容。”一个显示最近条目的pastebin列表就能很方便的完成这个功能，虽然对商业信息来说有点不怎么协调。

最后别忘了除了我们“设计”的部分之外，还有一个admin应用。按照我们在PasteAdmin类里设置的选项，它在admin里看起来应该如图10.5所示。

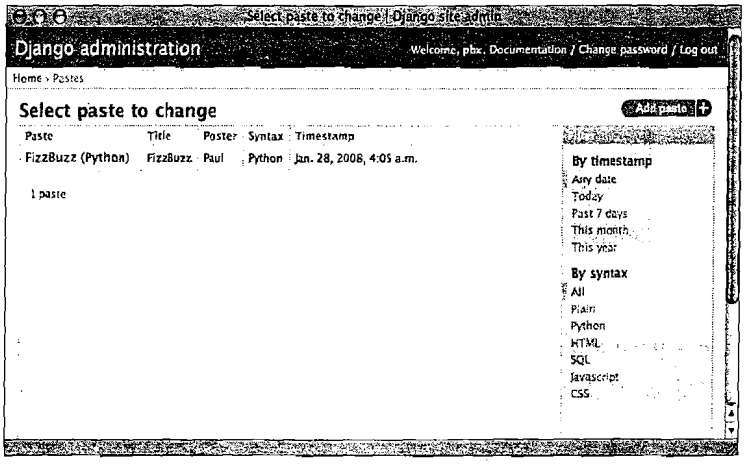


图10.5　admin界面

注意因为admin的list_display里第一个参数用的是模型的__unicode__方法而不是某个特定的模型变量，所以每一项可点击的名字都可以根据可用的消息来调整。

到此你已经完成了一个完全由Django通用视图驱动，可用并且好用的应用程序。虽然要再增强这个应用的比较合理的策略应该是开始使用自定义视图，不过你还是可以在保持通用视图的情况下做出很多改进的。这里就举三个例子：控制最近条目的列表，给代码加上语法高亮，另外还有定期清理旧的数据。

10.5 限制最近Paste显示的数量

Recently Pasted列表在条目不多的时候看着还可以，但是当这个网站变得非常忙碌的时候，列表很快就会变成庞然大物。要限制这个列表中的条目数量有很多种方法。因为我们这里用的是通用视图，所以管理它的最佳位置就是模板。

只需要在paste_list.html模板的第六行里稍微改一下代码，给object_list切片就行了。

```
{% for object in object_list|slice:":10" %}
```

它的工作原理是：URLconf向模板传递一个代表数据库里所有paste对象的queryset。根据models.py里的ordering = ('-timestamp')，它们都按照时间戳排好序。然后模板里的for循环取出列表里前十个结果并迭代之。

除了没有中括号以外，传给slice过滤器的值和我们传给一个普通Python对象的值是完全一

样的。所以Python里一个对应的例子就如下所示：

```
>>> number_list = [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> print number_list[:10]
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6]
```

如果Django的queryset不是“懒惰”的话（就是说我们要传递整个对象的列表然后除了前十个剩下的全部抛弃），那绝对不可想象。如果数目多达几千项，我们Web服务器进程的内存消耗就会直线飙升。但是因为queryset只有到最后一刻才会求值（在这里，就是模板里的for循环），所以在URLconf里指定Paste.objects().all()然后在模板里切片是不会引入任何负面影响的。而且，因为选择在列表里显示多少数目完全是显示层上的决定，所以再也没有比在模板里裁剪更好的地方了。

10.6 语法高亮

要是一个pastebin知道如何正确的给提交上来的代码应用语法高亮的话就会有用的多（也更有吸引力）。要做到这一点方法有很多（包括一个很棒的Python库Pygments，dpaste.com用的就是它），但是实现它最简单的途径就是在客户端里用JavaScript完成着色。

有一个叫Alex Gorbatchev的程序员用JavaScript创造了一个出色且广泛应用的着色工具。它的名字很简单，就叫Syntax Highlighter，你可以从Google Code上找到它（http://code.google.com/p/syntaxhighlighter/）。

在项目的网站上你可以找到完整的安装指导和各种使用Syntax Highlighter的示例，这里我们简单地介绍一下怎样给我们的Python代码加上语法着色。

首先，更新paste_detail.html模板如下。

```
{% extends "base.html" %}
{% block title %}{{ object }}{% endblock %}
{% block content %}
<h1>{{ object }}</h1>
<p>Syntax: {{ object.get_syntax_display }}<br>
Date: {{ object.timestamp|date:"r" }}</p>
<code><pre name="code" class="{{ object.get_syntax_display|lower }}">
    {{ object.content }}</pre></code>
<link type="text/css" rel="stylesheet"
    href="/static/css/SyntaxHighlighter.css"></link>
<script language="javascript" src="/static/js/shCore.js"></script>
<script language="javascript" src="/static/js/shBrushPython.js"></script>
<script language="javascript" src="/static/js/shBrushXml.js"></script>
<script language="javascript" src="/static/js/shBrushJscript.js"></script>
<script language="javascript" src="/static/js/shBrushSql.js"></script>
<script language="javascript" src="/static/js/shBrushCss.js"></script>
<script language="javascript">
dp.SyntaxHighlighter.HighlightAll('code');
</script>
{% endblock %}
```

我们给<pre>标签加上了name和class属性。这让我们的代码块可以在执行时被JavaScript找到。

```
<pre name="code" class="{{ object.get_syntax_display|lower }}">
```

这样就完成了。当浏览器渲染页面的时候，就会运行语法着色的JavaScript代码，在用户看到它们之前转换单色的代码。其输出结果如图10.6所示。

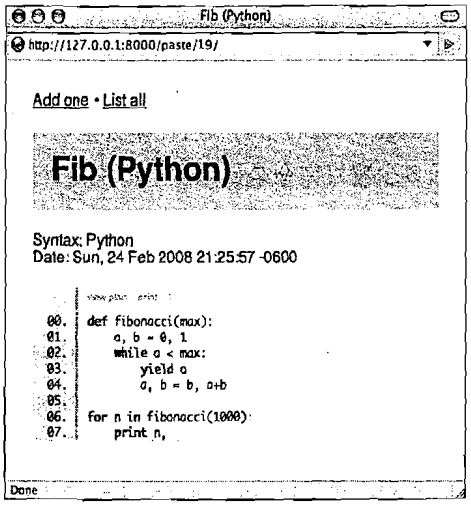


图10.6　经过着色后的Python代码

10.7　通过Cron Job清除

张贴到pastebin的代码通常都不会存在很久，所以最好能自动定期清理过旧的条目。而最好的方式当然就是利用服务器上每夜运行的cron job。

Cron job和其他用来在Web服务器环境之外运行的Django脚本都是Django的“就是Python”方式所展现出来的力量的又一极致体现。编写一个操作你的Django应用里的对象几乎不需要什么Django专用的东西。下面的脚本做出了以下的假设：

- 环境变量DJANGO_SETTINGS_MODULE被设置成包含指向项目设置文件的Python路径名的字符串（例如，“pastesite.settings”）。
- 在项目的settings模块里设置了EXPIRY_DAYS。
- 项目的名字是“pastesite”。

如果以上这些是正确的，那么除了测试和部署，你就不再需要做别的事情了。

```
#!/usr/bin/env python
import datetime
from django.conf import settings
from pastesite.pastebin.models import Paste

today = datetime.date.today()
cutoff = (today - datetime.timedelta(days=settings.EXPIRY_DAYS))
Paste.objects.filter(timestamp__lt=cutoff).delete()
```

脚本的最后一行就是关键所在——它使用Django的ORM来选择数据库里所有时间戳比计算出来的cutoff更老的对象，然后把它们全部删除。

> **注意**
>
> 根据数据库引擎的不同，你还可以定期“回收打扫”或重新声明被删除记录所留下的空间。你可以在http://djangosnippets.org上找到一段类似的操作SQLite数据库的代码段。

10.8 总结

希望到此你已经承认了Django通用视图的能力。我们的pastebin示例虽然实现的很简单，但是想想它具有的特性：输入验证，根据post重定向，细节和列表视图等。但更重要的是知道有了Django我们就等于给扩展打下了坚实的基础。如果我们要本地化应用，或是通过添加缓存以便承受Digg级别的流量，Django都能为我们做到。

第四部分 高级Django技术和特性

第11章 高级Django编程
第12章 高级Django部署

第11章 高级Django编程

在这一章里，我们要讨论一些可以应用到Django上的高级技术，例如生成RSS等聚合格式，自定义admin的行为，以及模板系统的一些高级用法。

下一章（第12章，“高级Django部署”）则是由几个相似的主题组合而成，包括了测试，数据导入，和编写脚本——都是和应用程序核心逻辑相互独立的功能。阅读这两章没有顺序上的要求——你可以按照自己的喜好跳跃着读。

11.1 自定义Admin

可以说，admin app是Django皇冠上的明珠。这种说法看着有点奇怪，一个在“contrib”目录里的组件能得到这么高的评价，毕竟这种目录的组件里通常都是可有可无的插件而绝非核心。不过Django工程师们的决定是正确的——在使用Django时并不一定要使用admin，所以它不应该是必须的组件。但是请不要忘记：admin app是一个非常强大和引人注目的应用程序。如果你的项目需要迅速创建一个漂亮实用的界面来添加编辑数据的话，admin绝对是不二之选。

在前面的内容里，你已经看到了如何通过ModelAdmin子类来进行一些定制的行为。比如，只需设置一下list_display，list_filter和search_fields选项就能满足你大多数的基本定制需要。

另外我们还提到，在以前例子中所使用的admin站点都是“默认的”admin站点——就是说你还可以同时设置多个不同的admin站点来获得更大的灵活性。比如，你可以让不同组的用户看到不同的admin界面。

不过最终admin还是不能满足你。经常会有Django的新手在使用admin一段时间后，说出这样的话来，“虽然我已经读过文档了，但就是这件事情我还是不知道怎么在admin里完成。要是加上这个功能的话，它就完美了！”

接下来，我们要向你展示一些定制以及扩展admin app的方法来完成人们经常提到的要求。在阅读的过程中请记住，admin其实就是一个Django应用而已。虽然它非常好用，界面也很漂亮，配置性也很强，但是如果最终它没办法满足你的要求时，你完全可以把它替换掉。

根据你打算要深入自定义的程度，有时候与其费尽心力把admin改的面目全非，还不如直接自己开发一个算了。如果现有的技术不适合你的话，完全可以考虑创建你自己的admin。除此之外，下面是admin里一些比较高级的定制应用。

通过Fieldsets改变布局和风格

admin app的fieldsets设置允许你细致的控制数据显示的方式。例如希望给某些特定元素加上CSS，按一定的方式组合变量，为选定变量添加JavaScript输入帮助，或是在初始状态下隐藏一些变量时，这个功能就很有用了。

以下是一个简单的模型例子，它通过fieldsets定制了显示的方式。

```
class Person(models.Model):
    firstname = models.CharField(max_length=50)
    lastname = models.CharField(max_length=50)
    city = models.CharField(max_length=50)
    state = models.CharField(max_length=2)

class PersonAdmin(admin.ModelAdmin):
    fieldsets = [
        ("Name", {"fields": ("firstname", "lastname")}),
        ("Location", {"fields": ("city", "state")})
    ]

admin.site.register(Person, PersonAdmin)
```

对Python来说，fieldsets设置就是一个包含两个元组的列表。其中第一个元组是这组变量的标题字符串，第二个元组则是对应这个组设置的字典。在字典里的键是选项的名字（稍后列出），对应的值则根据特定选项的不同而不同。在上面的例子中，变量选项被设置成一个变量名的元组。

这个例子的结果就是变量根据fieldsets里的设置组织在一起的显示结果，如图11.1。

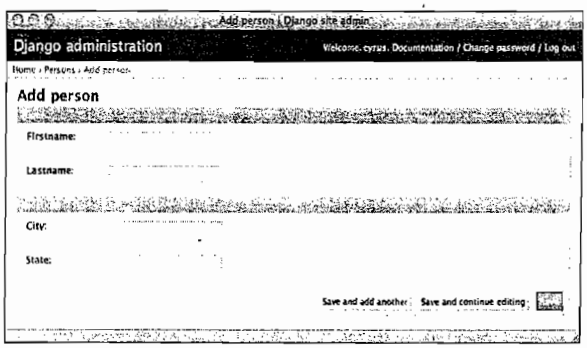


图11.1　admin里的新变量

以下是可以在fieldsets设置里使用的选项（就是用在前面字典里的键）。

- classes：这个选项指定了一个字符串元组，代表了应该应用到这个变量组上的CSS类名（在渲染模板里指定）。方便起见，admin事先定义了几个可以直接用的classes："collapsed"会让fieldset在标题下面折叠起来，可以通过基于JavaScript的开关展开；

"monospace" 则可用于HTML的textarea显示代码；另外还有"wide"，它让fieldset在admin里获得更大的宽度（虽然是固定的）。如果你想知道还有什么默认风格的classes可以使用的话，可以去看一下django.contrib.admin里的"media"目录。

> **注意**
>
> 如果你采用的是JavaScript“渐进增强”（progressive enhancement）的开发方式的话，通过选项添加的CSS类则正好为你自定义JavaScript代码提供了接口（假如你要给textarea添加一个所见即所得（WYSIWYG）编辑器的话）。

- description：该选项指定了一个用于描述field组的字符串。你可以把它想象成一个组级别的help_text。Django admin仔细编写了这部分默认的CSS风格，并将它清楚的显示出来。这种设计更为类似这样的简单标题增色不少。
- fields：前面说到，该选项指定了一组要在admin里显示在一起的变量名。如果选项里的字典有标注字符串的话（比如这个例子里的"Name"），那么它就会被当作该组的标题。否则该组就不显示任何标题。

扩展基本模板

除了修改简单的设置，你还可以通过替换一到多个基础模板来大幅调整admin app的外观。当然，实际上你并没有替换掉它，只不过是用你自己的文件重写了原本的模板，我们下面会给出解释。

本质上admin就是一个Django的应用程序，但是从各个方面来说它都是相当复杂的。它错综复杂的模板初学者通常未能架驭。比较合适的入门模板是base_site.html。以下是它的代码（稍作整理以便展示）。

```
{% extends "admin/base.html" %}
{% load i18n %}
{% block title %}{{ title|escape }}|{% trans 'Django site admin' %}{% endblock %}
{% block branding %}
<h1 id="site-name">{% trans 'Django administration' %}</h1>
{% endblock %}
{% block nav-global %}{% endblock %}
```

根据你对模板系统的知识，你一定猜到了开头的extends标签已经在背后为你做了很多工作。没错，真正在所有admin页面幕后的是一个更加复杂的模板 base.html。不过其中两个你要定制的东西（页面的<title>和<h1>标题）已经被抽象到 base_site.html里了。

只需把“Django site admin”和“Django administration”改成任何适合你应用需要的值然后保存模板就可以完成定制了。但问题是要把它保存到哪里去呢？

要覆盖这个模板的内容，你需要创建自己的拷贝，然后帮助Django模板加载器在默认模板之前找到它。你可以把这个文件放在你应用程序里的template目录下（如果你使用的是app_directories模板加载器），或者放在TEMPLATE_DIRS里的某个位置下。

不管是选择哪个方法，它都要被放在一个admin子目录里，这样其他admin模板里的extends标签才能正确定位到它。

如果你选择把它放在应用程序里（例如 myproject/myapp/templates/admin/base_site.html），请确保在INSTALLED_APPS里这个应用程序出现的顺序要在django.contrib.admin之前。因为模板加载器是按顺序搜索应用程序模板目录的，如果admin出现在你的应用程序的前面，那Django模板加载器在找到原来的那个模板后会停止搜索。

如果你还想继续深入探察admin模板的定制的话，下一个目标就是index.html。它比base_site.html要复杂一点，但是如果你需要重新布置admin主页，这个文件就值得好好研究。

添加新视图

除了重写整个admin app以外，自定义admin的终极步骤就是编写自己的admin视图。这比直接修改Django admin app的代码要好得多。

至少这种修改在每次Django的主代码进行更新的时候，都会给你自己以及写这些hack的人增加一个很麻烦的维护负担。如果来自Django的更新代码和你自定义的部分发生冲突，你就必须不断地去合并它们。相反，如果你的定制是以一个独立的admin视图形式完成的话，它们就很容易在Django程序员之间共享，他们还会随之将补丁和改进反馈给你。

编写新admin视图基本上是一件“Django”工作，和admin其实关系不是很大。这里有三个基本的要求：

- 视图URL最好是映射到admin app的URL空间“里面”去。虽然这更多是为用户的方便考虑，不过Django的URLconf系统让它变得很简单，所以没什么理由不这么做。
- 视图应该和admin的其他部分风格一致，或者说它应该用admin site模板来渲染它的响应结果。
- 视图应该使用Django的认证装饰器（下一节的内容）来控制访问。

认证装饰器

在这一章开始之前，如果你是新学Python并且需要复习一下装饰器（decorator）的概念的话，请在读过这一段后把书翻到第1章里关于装饰器的小节。装饰器是一种改变其他函数的函数。Django视图也是函数。你希望能用特定的方式来修改它们——即只有特定用户才能调用它们来限制访问——而装饰器非常适合这项工作。

你可以把装饰器看作是视图函数的守卫。如果用户通过了测试，他们就能进入——运行视图函数并显示页面。如果用户通不过测试，他/她就会被转至别处。（默认情况下，用户都会被重定向到 /account/login/ 并加上一个参数告知服务器，如果他们成功登录的话，再把他们重定向回原页面，比如 /account/login/?next=/jobs/101/。如果有需要，你还可以在settings.py里设置LOGIN_URL来自定义 URL。）

一个user_passes_test装饰器为你完成了大部分的执行认证工作。

除了要在你保护的视图上使用装饰器，你还需要提供一个工具函数来进行实际的工作。这

个函数必须接受一个User对象并且返回一个布尔值（True或False）。这种函数不一定要很复杂，甚至可以很简单：

```
def user_is_staff(user):
    return user.is_staff
```

有了函数定义，就可以这样使用装饰器：

```
@user_passes_test(user_is_staff)
def for_staff_eyes_only(request):
    print "Next secret staff meeting date: November 14th"
```

为user_is_staff还要专门定义一个函数，是不是太浪费了？这时可以用lambda（如果不知道lambda是什么，请参见第1章）。有了lambda，我们可以直接在装饰器里定义"匿名"函数。

```
@user_passes_test(lambda u: u.is_staff)
def for_staff_eyes_only(request):
    ...
```

回到我们的例子上来，假设你需要比“是或不是staff”更好的分类。Django的admin和auth app提供了非常方便的权限系统来让你管理用户。你知道每个模型在admin里都有它自己的创建、更新、删除权限，不过在Python代码里也是可以访问这些权限的。比如说你现在有一个SecretMeeting模型以及对应在auth系统里的权限，如果你打算创建一个视图来允许特定的用户添加新的 secret meeting的话，相应的装饰器函数就应该如下所示：

```
@user_passes_test(lambda u: u.can_create_secretmeeting)
def secret_meeting_create(request):
    ...
```

通过配合使用Django预定义的权限和应用相关的代码，你可以按照任何你喜欢的方式控制访问。

11.2 使用聚合

Django提供了一个非常方便的工具，Syndication app（django.contrib.syndication）可以在任何模型对象的集合上生成RSS或是Atom feeds。这极大简化了为Django项目加入聚合功能的工作。

Syndication app的配置自由度非常高，不过只使用默认值就能轻松地开始使用它了。这里只有两个必须的步骤：首先是一个生成feed对象的特殊类；其次是在 URLconf里的规则，它规定了如何将这些对象传递给Syndication app。理解它的工作方式最简单的办法就是直接看代码。以下例子是基于第2章中开发的Blog应用之上。

Feed类

你可以通过定义一个包含特殊feed类的单独模块来配置feed，这个模块随后会被直接导入

到URLconf里。Django没有规定这个文件的位置（甚至叫什么名字也无所谓），但是比较常见的做法是放在应用程序目录下，同时把它命名为feeds.py。所以在你的mysite/blog目录里，创建一个包含如下代码的feeds.py文件：

```
from django.contrib.syndication.feeds import Feed
from mysite.blog.models import BlogPost
class RSSFeed(Feed):
    title = "My awesome blog feed"
    description = "The latest from my awesome blog"
    link = "/blog/"
    item_link = link

    def items(self):
        return BlogPost.objects.all()[:10]
```

以上title和description属性是给feed的用户（例如，一个桌面RSS阅读器）用来标识feed的。link属性指定了什么页面和feed相关联。站点的域名则是由你提供的局部URL决定的。

item_link 属性决定了阅读器要查看某个单独feed项的Web页面时载入的页面。item_link设置为 /blog/的意思是点击任何单独feed项都会把用户转去blog首页——要是你觉得这有点不灵活的话（确实是很不灵活），更好的做法是为 BlogPost模型添加一个get_absolute_url方法，Syndication app会自动去调用它。

items方法是这个类的核心——它决定了要返回什么对象给Syndication app。这里返回的是blog帖子里的前十篇，因为BlogPost模型在Meta里把ordering属性设置为"-timestamp"（时间逆序），所以我们得到的是最新的十个帖子。

这个类只包含了Syndication app运行所必须具有的组成部分。其实还有很多可选的属性，以及更灵活的指定这些必需属中点的方法。理解feed类的很多属性其实都属于下面三个类别之一，对学习定制化很有帮助：

- 硬编码值（比如前面的title，description和link）
- 没有显式参数的方法（当然引用feed的self总是有的）
- 接受一个显式参数的方法（一个独立的feed项）

Syndication app会通过Python的"duck typing"智能的分辨出你在用的是哪一个类别。Syndication app会按照和上面列表的相反顺序查找这三个选项。如果发现了一个名字匹配且接受一个参数的方法，它就会将当前的feed对象传递给该方法；接着，查找无参的方法并调用之；如果前面两项都没找到，它就会去尝试查找硬编码。如果全部都失败了的话，它会抛出FeedDoesNotExist异常（这种情况代表你的feed是损坏的）。

为Feed指定一个URL

创建完能处理feed生成的类后，剩下的就是给这个feed安上一个可用的URL了。继续我们blog的例子，编辑更新URLconf文件（mysite/blog/urls.py）如下：

```
from django.conf.urls.defaults import *
from django.contrib.syndication.views import feed
from mysite.blog.views import archive
from mysite.blog.feeds import RSSFeed

urlpatterns = patterns('',
    url(r'^$', archive),
    url(r'^feeds/(?P<url>.*)/$', feed, {'feed_dict': {'rss': RSSFeed}}),
)
```

以上只添加了三行代码。其中前两行是import，我们从Syndication app里导入feed view对象，然后从blog app里导入RSSFeed类。

另外我们还添加了一个看起来很复杂的URL模式。它可以拆成三个部分：

- r'^feeds/(?P<url>.*) /$'：这是我们的URL正则表达式。因为我们是在“blog”应用内部，也就是在URL /blog/ 下，这意味着它会匹配 /blog/feeds/FOO/ 这样形式的URL，然后把"FOO"传递给视图函数。
- feed：这是我们从django.contrib.syndication.views里导入的视图函数。
- {'feed_dict':'{'rss':RSSFeed}}：在任何urlpattens元组里，我们都可以提供像这样的第三个参数，这个字典用来向视图函数传递更多的参数。这里传递的是一个叫feed_dict的参数，它是一个字典，里面只包含了一个元素，将字符串“rss”映射到RSSFeed类。如果要添加其他类型的feed，只需创建相应的类然后在这个字典里引用它们就可以了。

虽然在这个字典里的键（"rss"）正好是feed的类型，不过这个名字是可以任意取的——比如返回最新帖子的feed就可以取名为"latest"。Syndication app对此没有做任何限制。它只关心要怎么把请求和正确的Feed类联系起来。

再论Feeds

对一个定期（或不定期）更新的站点来说，提供RSS或Atom都是一大进步。虽然这些格式表面上看来挺简单，但是如果要自己动手写这些feed输出的话还是很麻烦和很容易出错的。著名的Universal Feed Parser（http://feedparser.org/）提供了超过三千个测试用例来确保它能稳定的处理各种类型的（或者说各种糟糕的）发布feed。而通过Django的Syndication app，你只需很少的代码和配置就能生成干净合法的输出。

如果这里介绍的自定义内容还是不能满足你的要求，请进一步查阅Django的官方文档。真正要你自己写的机会微乎基微。

11.3 生成下载文件

Django是一个Web框架，所以自然而然地我们会觉得它就是一个通过HTTP传输生成HTML的工具而已。但其实Django完全适用于各种类型的内容已及各种传输方式。

Django凭借的有两点。首先Django的模板语言是基于文本，而不是XML的。如果一个Web框架只能处理XML模板的话就会在生成纯文本报告或是email的时候捉襟见肘。

其次，Django驱动网站的HTTP机制允许你调整HTTP的包头信息。所以，你可以把Content-Type设置为非HTML类型，比如 JavaScript（和在第9章里的JSON视图一样）或者增加一个Content-Disposition头来强制下载。

这里是一些使用Django来生成非Web页面输出的例子。其中有些用到了模板系统，就像刚才说到的那样；其他的则没有使用模板，因为有时候用模板纯粹是给自己找麻烦。记住，用正确的工具做正确的事。

Nagios配置文件

首先和你分享一个现实世界里的例子，即著名的开源监视系统，Nagios（http://nagios.org/）。和许多类似的项目一样，它也遵循了Unix的惯例，采用纯文本配置文件作为发行格式。这种配置文件格式对Django的模板系统来说再好不过了。

在编写本书的时候，本书的作者之一正在编写一个小型的，高度定制的内部应用程序来（除了其他的）生成一部分Nagios配置。它作为一个Django应用部署在一个中心数据库系统上，用来提供系统和服务的信息。随后这份信息会导出成Nagios可以理解的格式，让用户能跟踪系统的某个部分并通过Nagios 监视它们。

以下是一个系统－服务层次模型示例的简化（在实际的应用程序里，这些信息被分散到多个模型中）。

```
class System(models.Model):
    name = models.CharField(max_length=100)
    ip = models.IPAddressField()

class Service(models.Model):
    name = models.CharField(max_length=100)
    system = models.ForeignKey(System)
    port = models.IntegerField()
```

下面的模板可以为每个主机的Nagios生成service-check文件，同样这也是一个简化过的版本。它需要的环境是一个System对象，system。

```
define host {
    use             generic-host
    host_name       {{ system.name }}
    address         {{ system.ip }}
}

{% for service in system.service_set.all %}
define service {
    use                     generic-service
    host_name               {{ system.name }}
    service_description     {{ service.name }}
    check_command           check_tcp!{{ service.port }}
}
```

> **注意**
>
> 不要被这里的单个大括号弄糊涂了，它们是Nagios文件格式的一部分，不会被Django模板系统解析。模板系统只关心两个大括号或是大括号加上百分号。

上面的模板在渲染后会返回给我们一个可用的Nagios文件，它将system定义为Nagios主机，同时输出任何与之相关的服务，并用一个网络检查命令来确认记录的TCP端口正在运行中。

vCard

vCard 格式是一个基于文本的名片格式。很多流行的地址簿，email和PIM（Personal Information Manager，个人信息管理器）产品都支持导入导出这种格式，包括Evolution， The OS X Address Book和Microsoft Outlook。

如果你的Django应用包含联系人信息的话，你会希望能为它生成vCard格式，这样用户就可以把那些信息导入到本地的地址簿或是PIM应用程序里了。

下面这段代码用到了vObject模块（可以在 http://vobject.skyhouseconsulting.com 下载到），它能简单地生成vCard数据。

```
import vobject

def vcard(request, person):
    v = vobject.vCard()
    v.add('n')
    v.n.value = vobject.vcard.Name(family=person.lastname, given=person.firstname)
    v.add('fn')
    v.fn.value = "%s %s" % (person.firstname, person.lastname)
    v.add('email')
    v.email.value = person.email
    output = v.serialize()
    filename = "%s%s.vcf" % (person.firstname, person.lastname)
    response = HttpResponse(output, mimetype="text/x-vCard")
    response['Content-Disposition'] = 'attachment; filename=%s' % filename
    return response
```

这里最重要的部分就是修改HttpResponse对象来提供一个非HTML的MIME类型和文件下载。我们没有直接返回HttpResponse，而是先创建了这个对象，设置其内容和MIME类型。然后为响应设置Content-Disposition头来指定一个附件。在视图最后返回这个响应对象就能启动传输过程，让用户接受生成的文件。

这项技术展示了Django在混合各级工具方面的出色的能力。HttpResponse对象能简单地让我们利用浏览器，服务器和HTTP的知识来定制要返回的响应——同时不需要考虑响应里无关的其他部分。

这种方法可以用来处理任何你希望生成和传输的非HTML文件类型，下面是另外一个例子。

Comma-Separated Value (CSV)

如果你的应用程序要输出表格化的数据，CSV（comma-separated value）格式绝对是最简单实用的。所有的编程语言都可以处理这种格式，而且几乎所有能处理格式化数据的应用程序（Microsoft Execel，Filemaker Pro）都可以导入导出它。Python也提供了一个csv模块来帮助你处理这种数据格式。

很多程序员在一开始碰到CSV这类任务的时候都会尝试自己编写解析器和生成器。毕竟CSV这种格式太简单了，对吧？它就是一些逗号加上一些数字，可能还有一点字符。要是这些字符里包括逗号的话可能还会用到引号。要是引号里还有引号的话可能还会用到转义符。开始有点复杂了吧。好在已经有前人帮我们编写好模块来处理它。

假设现在我们要把人名数据从刚才的vCard转换到一个简单的数据表里，其中每一列分别是名字，姓氏和email地址。首先，我们先在解释器里实验一下确保csv模块工作正常。因为csv模块主要是设计用来处理文件类的对象，所以这里我们用Python方便的 StringIO模块来捕捉输出。（StringIO为字符串提供了一个模仿文件的接口。）

```
>>> import csv, StringIO
>>> output = StringIO.StringIO()
>>> output_writer = csv.writer(output)
>>> people = [("Bob", "Dobbs", "bob@example.edu"),
... ("Pat", "Patson", "pat@example.org"),
... ("O,RLY", "O'Reilly", "orly@example.com")]
>>> for p in people:
...     output_writer.writerow(p)
...
>>> print output.getvalue()
Bob,Dobbs,bob@example.edu
Pat,Patson,pat@example.org
"O,RLY",O'Reilly,orly@example.com
```

看起来还不错——它自动给包含逗号的项加上了引号。那么这跟Django视图函数有什么关系呢？多亏Django的HttpResponse和StringIO对象一样也是一种" 类似文件"的对象，所以它们很相像——就是说它们都提供csv.writer需要的write方法。

```
def csv_file(request, people):
    import csv
    response = HttpResponse(mimetype="text/csv")
    response_writer = csv.writer(response)
    for p in people:
        response_writer.writerow(p)
    response['Content-Disposition'] = 'attachment; filename=everybody.csv'
    return response
```

当用户请求访问到这个视图的时候，还是会返回一个HttpResponse对象（如果你还记得的话，所有的Django视图都要求这一点），但是它包含的内容将是text/csv，而非text/html，并且大多数浏览器在遇到Content-Disposition头时，都会触发下载而不会把它显示在浏览器里。

CSV对解决“能不能从Django获取数据导入到X应用里”这种问题是很方便的。几乎任何处理结构化数据的程序都能读入CSV。

用PyCha实现图表

PyCha（http://www.lorenzogil.com /projects/pycha/）是Python里一个简单优雅的图表库，构建在Cairo图形系统之上。PyCha并没有试图容纳所有的输出格式或者配置选项，但是它具备了两个优点：相对简单的Pythonic语法以及外观美好的默认输出显示。

在上面CSV的例子里，Django特有的一个重要技巧就是把输出放到一个字符串里，这样我们就可以在响应里设置正确的mime-type并返回它。

```
def pie_chart(request):
    import sys, cairo, pycha.pie
    data = (
        ('Io on Eyedrops', 61),
    ('Haskell on Hovercrafts', 276),
    ('Lua on Linseed Oil', 99),
    ('Django', 1000),
    )
dataset = [(item[0], [[0, item[1]]]) for item in data]
surface = cairo.ImageSurface(cairo.FORMAT_ARGB32, 750, 450)
ticks = [dict(v=i, label=d[0]) for i, d in enumerate(data)]
options = {
    'axis': {'x': {'ticks': ticks}},
    'legend': {'hide': True}
}
chart = pycha.pie.PieChart(surface, options)
chart.addDataset(dataset)
chart.render()
response = HttpResponse(mimetime="image/png")
surface.write_to_png(response)
response['Content-Disposition'] = 'attachment; filename=pie.png'
return response
```

这里的关键是获取了一个surface对象（这是Cairo里最主要的绘制元素），并且通过把它传递给pycha.pie.PieChart构造函数来把 chart画到surface上面，然后再调用chart的render函数。当绘制完成后，再用surface的write_to_png方法把二进制 PNG数据写到一个文件对象里并把它返回给响应。

所以视图函数最后返回的是一张PNG图片，我们只要设置好相应的包头并把它发送给浏览器进行下载就可以了。

图11.2就是生成的图片结果。

当然了，在实际使用视图的时候，我们不会把数据硬编码在视图函数里，而是会通过Django的ORM查询从数据库里获取它们。根据应用程序的不同，你甚至可以考虑编写一个自定义的模板标签来专门从特定的QuerySet里获取数据，然后把它渲染到PNG图表里去。不过由于即时生成图片文件很消耗资源，这一过程最好使用缓存。

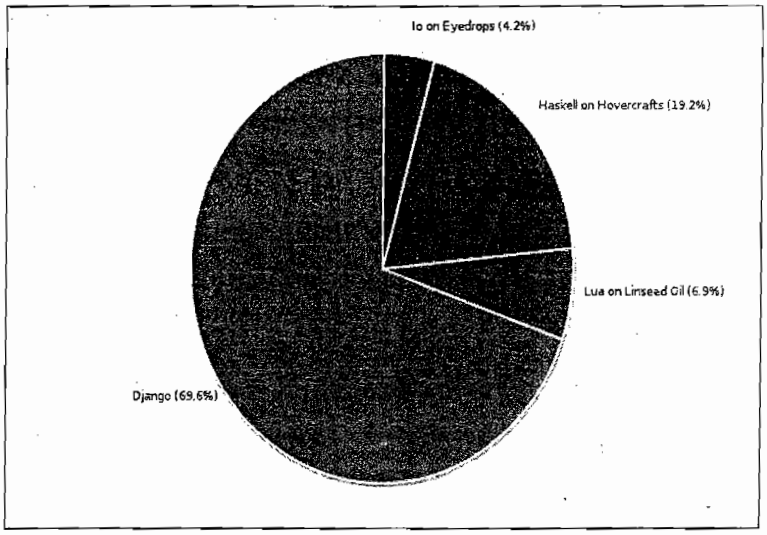


图11.2　PyCha饼图示例

11.4　用自定义Manager来增强Django ORM

虽然Django的ORM系统没有要完全替代SQL的意思，不过对大多数Web应用来说它都绰绰有余了。而且你还学习了如何通过额外的方法直接使用SQL命令来辅助Django ORM查询。不过，自定义manager给你提供了另一种途径。可能你还不知道其实你已经在下面这样的代码里用到了manager：

```
really_good_posts = Post.objects.filter(gemlike=True)
```

这里Post.objects就是一个manager对象，它是一个models.Manager子类的实例。这个类的方法决定了你能对一个queryset进行的操作——比如这个例子里的过滤。

一个自定义manager也是一个由你自己定义的models.Manager子类。它主要有两个用途：修改默认返回的对象集合（一般是数据表里全部的对象）以及添加操作这个对象集合的新方法。

修改默认对象集

如果不想一遍又一遍的写下前面的查询代码，而希望用简洁一点的方法来告诉Django你只要数据库里“gem-like”的帖子该怎么做？很简单，写一个象以下这样的manager就可以了：

```
class GemManager(models.Manager):
    def get_query_set(self):
        superclass = super(GemManager, self)
        return superclass.get_query_set().filter(gemlike=True)
```

这个类要怎么用呢？只需要把它赋值为你模型（model）里的一个属性就行了。

```
class Post(models.Model):
    """My example Post class"""
    title = CharField(max_length=100)
```

```
    content = TextField()
    gemlike = BooleanField(default=False)

    objects = models.Manager()
    gems = GemManager()
```

注意现在你的模型里实际上有两个manager——一个是objects，另一个是gems。它们都能返回queryset。objects的作用和默认manager是一样的。

> **注意**
>
> 在这里定义的objects manager和通常Django自动为我们创建的manager是一模一样的。在这个模型里我们必须显式定义它的原因在于当Django有额外 manager出现时的行为——它看到的第一个manager会当成模型的默认manager，admin会在选择对象时使用它。如果我们省略 objects = models.Manager() 这一行的话，admin就无法再显示“非gem-like”的帖子了。

看过gems这个新的manager的代码之后，它的作用大概你就了然于心了。它通过Python的super函数去调用父类（models.Manager）的get_query_set方法，然后过滤结果，就像之前我们在自定义的manager里做的那样。

那要怎么用它呢？你可能已经猜到了——和默认的objects manager一样。

```
really_good_posts = Post.gems.all()
```

而且由于这个新的自定义manager返回的也是一个queryset，你还可以进一步对它进行过滤，排除，以及其他queryset方法的操作。

添加新的Manager方法

之前定义的自定义manager算是一种很有用的语法糖。就是说它可以把一些本来比较罗嗦的查询：

```
really_good_posts = Post.objects.filter(gemlike=True)
```

变成更简洁的形式：

```
really_good_posts = Post.gems()
```

如果要处理很多这类对象集的话，明显能让我们的代码更加易读，但是这还不是自定义manager的全部能力。通过为它添加自定义方法允许参数传递，我们能获得更大的灵活性。

继续刚才的例子，假设我们要简洁地指定一个queryset，它只包含了在标题和正文里提到特定单词的帖子。在一般的Django语法里，我们可以这么写：

```
cat_posts = Post.objects.filter(gemlike=True, title__contains="cats",
    content__contains="cats")
```

这个查询显得稍微长了一点，要是这种查询会重复使用的话，我们希望用更简明的方式，比如：

```
cat_posts = Post.objects.all_about("cats")
```

> **注意**
>
> 创建类似的类和方法时最有意思的挑战之一大概就是给它们起一个好名字。要找到一个能在Post.objects.foo这样的形式里读起来顺口的名字可不容易，毕竟你（或任何会阅读你代码的人）都会经常看到它。一般来说，manager应该是一个名词（“objects”，“gems”），而manager方法则应该是动词（“exclude”，“filter”）或是形容词（“all”，“latest”，“blessed”）。

以下是一个manager方法的代码：

```
class AllAboutManager(models.Manager):
    """Only return posts that are really good and all about X"""
    def all_about(self, text):
        posts = super(AllAboutManager, self).get_query_set().filter(
            gemlike=True,
            title__contains=text,
            content__contains=text)
        return posts
```

严格说来，自定义manager并没有提供额外的功能。但是模型的API接口越是干净整齐，项目里使用它的其他代码就越容易编写和维护。

11.5　扩展模板系统

在第6章，“模板和表单处理”中我们讲到了Django的模板包含了一些经过深思熟虑的设计，这在同类系统里是没有的。它的设计可以产生各种类型的文本，而不局限于XML等变型，比如XHTML。这在生成JavaScript、CSS，纯文本E-mail等一系列文本输出的时候非常有用。

Django模板和其他系统另一个不同的地方在于它没有重新发明，或是封装一个完整的编程语言。这加快了模板处理的速度，减少了框架整体的复杂度，同时对非程序员背景的网页设计师来说也能尽量保持简洁。

内置的模板系统通常已经足以应付多数项目的使用。但是有时候你还需要或希望它更强大一点。在这一节里，你会学习到如何创建自定义模板标签和过滤器，甚至用第三方的模板系统替换掉Django的默认模板。

简单的自定义模板标签

假设你要在网站首页上显示一幅随机选择的图片。以你现在的知识水平，要做到这一点非常容易。你只需在视图代码里构建一个图片文件的列表，然后用Python的random.choice函数为你选中一幅，将它的值传给模板就可以了。这个视图函数看起来如下所示：

```
def home_view(request):
    img_files = os.listdir(settings.RANDOM_IMG_DIR)
    img_name = random.choice(img_files)
    img_src = os.path.join(settings.RANDOM_IMG_DIR, img_name)
```

```
# ... other view processing goes here ...
render_to_response("home.html", {'img_src': img_src})
```

（这段代码遵循了Django保存配置信息的推荐做法，例如把RANDOM_IMG_DIR放在settings.py文件里，这样项目里所有的应用都可以访问它，修改起来也方便。）

最后，在模板里用一个图片标签就能获取传递进来的值了。

```
<img src="{{ img_src }}" />
```

如果只是这样的话那一切都很顺利。但假设你决定要把这个随机图片加到由另一个视图驱动的页面里去。或者你的设计师说道，“嘿，我有五个要用到这个随机图片功能的地方，其中三个要从不同的目录里读入图片，你有没有办法呀？”当然，答案是没问题！

这里的关键是一种叫做自定义模板标签（custom template tag）的特性。Django用来定义它自己的模板标签的机制，作为程序员的你当然也可以一样使用。也就是说你可以为你的设计师创造一个简单的标签。更棒的是，标签还可以接受路径参数，所以你的设计师还能拥有他/她自己的“专用”版本。

这类标签的代码相当简单，所以我们先来看实现，然后再返回来解释细节。这个标签最后应该能够做到以下这样的事：

```
<img src="{% random_image "faces/thumbnails" %}" />
```

标签的名字是random_image，跟在后面加引号的字符串是路径名。这是相对于MEDIA_ROOT的相对路径。Django模板系统会解析参数然后把它传递给你的函数（模板标签就是函数）。

以下是定义标签的完整代码。和往常一样，我们可以导入任何需要的模块。这里大部分代码都是纯Python，但我们主要介绍Django相关的部分。

```
import os
import random
import posixpath
from django import template
from django.conf import settings

register = template.Library()

def files(path, types=[".jpg", ".jpeg", ".png", ".gif"]):
    fullpath = os.path.join(settings.MEDIA_ROOT, path)
    return [f for f in os.listdir(fullpath) if os.path.splitext(f)[1] in types]

@register.simple_tag
def random_image(path):
    pick = random.choice(files(path))
    return posixpath.join(settings.MEDIA_URL, path, pick)
```

第一个值得注意的就是 register = template.Library() 这一行。template.Library实例允许我们访问一些特定的装饰器，它们可以把函数转换成模板系统可以使用的标签和过滤器。虽然这个

实例的名字可以任取，Django还是强烈推荐你使用register这个名字，这让其他人能更容易理解你的代码。

files函数是一个简单的帮助函数，提供给我们一个用扩展名标识出来的图片文件列表。

random_image 函数会在模板使用标签的时候执行，其path参数也是来自模板标签。它用files函数得到图片文件名的列表，随机选中一个，加上MEDIA_URL前缀来生成一个适合img标签使用的路径，然后返回它。(这里的posixpath.join和POSIX路径没有任何关系，只不过这个函数在拼接URL的时候很好用而已，它能保证每个部分之间只有一个斜杠，而且和os.path.join不同，即便是在Windows系统上它还是会使用斜杠而不是反斜杠。)

如果说这段代码里有什么神奇的地方，那一定就是在random_image函数上的@register.simple_tag装饰器了。正是这个装饰器把我们的简单函数转换成模板引擎可以使用的标签。

虽然这里我们只定义了一个标签，但实际上我们创建的这个文件算是一个能包含很多标签的Django标签库。所以在保存文件的时候记得给它起个好名字，比如将来我们会往这个库里添加其他随机内容标签的话，就可以叫它randomizers.py。

这个文件需要被保存在一个叫做templatetags的目录里，并且这个目录要在模板系统的搜索路径上。即要么是在INSTALLED_APPS里（如果你在settings.TEMPLATE_LOADERS里有列出django.template.loaders.app_directories.load_template_source的话），要么是在TEMPLATE_DIRS设置里列出的目录之一（如果你在settings.TEMPLATE_LOADERS里有列出django.template.loaders.filesystem.load_template_source的话）。

这个templatetags目录必须是一个Python的包，也就是说在目录下面要放一个__init__.py文件（空的就行）。要是你忘记创建这个__init__.py文件的话，你会得到一个对该问题没任何帮助的错误信息。

```
TemplateSyntaxError at /yourproject/
'randomizers' is not a valid tag library: Could not load template library
from django.templatetags.randomizers, No module named randomizers
```

所以要是你看到这样的错误，别忘了到你的templatetags目录下去创建该文件。(要是你需要复习一下为什么Python需要这个文件的话，请参考第1章里的模块和包一节。)

完成之后，你的新tag就可以用在项目里的任何应用程序里了。

要在给定模板里使用这个新标签，你需要在模板顶部使用模板系统的load标签来加载它。load标签只需要一个参数，即库的模块名（不用".py"）。

```
{% load randomizers %}
```

标签库载入模板后，它定义的标签就可以和Django的内置标签一样使用了。你的新random_image标签既可以接受一个字符串，也可以接受一个模板变量作为其参数。所以你可以在视图里动态指定要获取随机图片的目录，然后把它作为image_dir传递给模板。这样，random_image标签的使用看起来就如下所示：

```
<img src="{% random_image image_dir %}" />
```

要定义接受多个参数的标签也非常容易。simple_tag装饰器和Django的模板系统会帮你检查传递进来的参数数目和期望的是否一样。

现在我们在原有的基础上再来创建一个新的随机图片标签，这次它要接受第二个参数。假设我们要明确地指定所需的文件类型（扩展名）。比如我们的图片目录里包含有PNG，GIF和JPEG文件，而我们只想从PNG文件里选择。

以下是要加入到randomizers.py文件里的新函数。

```
@register.simple_tag
def random_file(path, ext):
    pick = random.choice(files(path, ["." + ext]))
    return posixpath.join(settings.MEDIA_URL, path, pick)
```

这个新标签叫做random_file，它也调用了前一个版本里用到的files函数。这个新版本添加了第二个参数ext，并把它（加上一个“.”前缀后作为一个单元素列表）传递给files函数作为其可选的第二个参数。

新的模板标签的用法就变成如下所示：

```
<img src="{% random_file "special/icons" "png" %}" />
```

对于希望给模板的作者（也包括你自己）提供一个简洁可读的方式来生成原本需要在多个视图函数里通过自写代码才能生成的值的情况，简单的自定义模板标签就是最适合的办法。如果你还需要更复杂的方式，请往下读。

Inclusion标签

在上面的例子中标签返回的是简单的字符串。如果你希望自定义标签返回更复杂的内容（比如一段动态的HTML），千万不要用模板标签函数来构建和返回该HTML，即使这个做法看起来挺不错。MVC原则（见第3章）的意思就是要将HTML放在模板里，而不应该放在视图函数里，同样，你也应该尽量 避免在模板标签函数里硬编码HTML。

虽然你可以写一个simple_tag来使用模板引擎，不过Django提供了更方便灵活的方法：inclusion标签。

Inclusion标签在你要用当前模板context里的一些值来渲染一块内容时是最有用的。例如，假设你的模板里有一个包含了当前日期的{{ date }}变量，然后你想用一个模板标签来为当前的月份渲染出一个简单的月历。

现在我们来编写这个标签。这个标签是基于Python的calendar模块，这个模块会把月份作为一个包含了星期数字列表的列表返回给我们（在月份开始和结束前后的星期数字都用0填充）。

```
>>> import calendar
>>> calendar.monthcalendar(2010, 7)
[[0, 0, 0, 1, 2, 3, 4], [5, 6, 7, 8, 9, 10, 11], [12, 13, 14, 15...
```

现在我们要把列表里的0都变成空白——因为我们不想让月份开始和结束前后的日子显示为零。在calendar.monthcalendar上用列表推导式就能得到我们想要的结果了。

```
>>> import calendar
>>> month = calendar.monthcalendar(2010, 7)
>>> [[day or '' for day in week] for week in month]
[['', '', '', 1, 2, 3, 4], [5, 6, 7, 8, 9, 10, 11], [12, 13, 14...
```

这个巧妙的列表推导式的意思是遍历月份里的每个星期，对每个星期遍历其中的每一天，如果这一天非零，那么就直接使用它，否则就使用空字符串。

由于Django的模板引擎并不关心我们传递的是整数还是字符串，所以这里混杂的数据类型不会给我们造成困扰。不过要是我们把这个数据传给Python函数做进一步处理的话，这种混合数据就不能直接用了。

calendar模块甚至还为我们提供了日期和月份的名字。

```
>>> list(calendar.day_abbr)
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
>>> list(calendar.month_name)
['', 'January', 'February', 'March', 'April', 'May', 'June', 'July'...
```

在现代CSS布局里，月历还是属于表格型数据，所以我们用HTML的表格标签来生成它。我们来为这个月历创建一个小型的模板，这里假设月历是按星期传入的完整列表：

```
<table>
<tr><th colspan='7'>{{ month }} {{ year }}</th></tr>
<tr>{% for dayname in daynames %}<th>{{ dayname }}</th>{% endfor %}</tr>
{% for week in weeks %}
    <tr>
    {% for day in week %}
        <td>{{ day }}</td>
    {% endfor %}
    </tr>
{% endfor %}
</table>
```

对于只作为模板一部分的模板，包括这里的用作inclusion标签的模板，我们都建议在命名前面加上一个下划线。注意它们不是要被用作完整文档框架的。给你的模板片段取名为_canlendar.html并把它保存在模板系统可以找到的地方，比如应用程序的templates目录或是TEMPLATE_DIRS里的目录之一都行。

现在我们来创建实际的inclusion标签函数。在templatetags目录里新建一个文件，保存为inclusion_tags.py。

```
@register.inclusion_tag("_calendar.html")
def calendar_table():
    import calendar
    import datetime
    date = datetime.date.today()
    month = calendar.monthcalendar(date.year, date.month)
    weeks = [[day or '' for day in week] for week in month]
    return {
        'month': calendar.month_name[date.month],
```

```
        'year': date.year,
        'weeks': weeks,
        'daynames': calendar.day_abbr,
    }
```

注意我们的函数返回的是一个字典，它会在渲染模板的时候作为context字典提供给@register.inclusion_tag来调用。换句话说，字典里所有的键都变成模板里的变量，可以用来显示相应的值。

除了这里说到的内容，inclusion标签并没有其他特别之处。它们用来帮助你方便地将表现和逻辑层分开。你当然可以用simple_tag来创建一个返回巨大HTML的标签，但是那样的话非常难以维护。或者也可以创建一个标签，让它自己去调用模板引擎，不过这样又未免太繁琐。Inclusion标签让你既能编写简洁的代码，同时又能把模板的内容保持在模板内部即它本该所在的地方。

现在我们给页面加上一个简易的样式表来稍微装点一下这个日历。将下面的代码放到_calendar.html文件的顶部。

```
<style type="text/css">
td, th { padding: 4px; width: 30px; background: #bbb; }
td { text-align: right; }
</style>
```

当你要使用这个标签的时候，只需要在模板最开始的地方输入{% load inclusion_tags %}即可。具体调用的时候，在我们希望日历出现的地方输入{% calendar_table %}就行了。

这个日历看起来应该如图11.3所示（根据浏览器的不同可能会稍有变化）。



图11.3 我们的日历

显示效果看起来还不错。由于商业逻辑和表现层区分的很清楚，你的设计师可以轻易的调整标签的模板文件。同时，你和其他程序员可以对内容作出修改，比如本地化月份和日期的名字，而根本不用去碰模板（或是它包含的任何其他模板）。

自定义过滤器

Django的模板系统自带了很多非常有用的过滤器（filter），但是有时候你会需要添加自己的过滤器。过滤器很容易使用，实际上编写起来也不难。和其他你看到的标签一样，它用到的是方便的装饰器语法。

过滤器就是函数——在大多数情况下接受字符串并返回字符串的函数。一个不太复杂又很

有趣的例子（至少不是太长，要是你不喜欢正则表达式的话可能还看着有点晕）就是一个将驼峰状（CamelCase）的词组转换成适合HTML使用的wiki链接的“wiki化”过滤器。如下所示：

```
import re
from django.template import Library
from django.conf import settings

register = Library()
wikifier = re.compile(r'\b([A-Z]+[a-z]+){2,}\b')

@register.filter
def wikify(text):
    return wikifier.sub(r'<a href="/wiki/\1/">\1</a>', text)
```

这个标签接受一个字符串，字符串里所有驼峰状的词组都被替换成 /wiki/CamelCaseWord/ 这样的链接。它可以用在以下这样的模板里（假设变量title和content分别保存了页面的标题和wiki标记的内容）。

```
{% load wikitags %}
<h1>My Amazing Wiki Page: {{ title }}</h1>
<div class="wikicontent">
{{ content|wikify }}
</div>
```

拥有额外参数的过滤器

上面这个wiki化的函数接受一个参数，它的值是模板里出现在过滤器前面"|"（管道）符号左边表达式的值。不过要是你希望过滤器的用户还可以通过传递一个额外的参数来调整过滤器的行为的话要怎么办？

过滤器当然可以接受第二个参数了。通常该参数可以用来调整过滤器的行为。例如，假设你想让一个显示字符串的函数只在字符串包含特定的字符序列时才显示它（想象一下Unix命令grep）。当然用if/then模板标签也能做到，不过过滤器的方式更加简洁。

```
{{ my_string|grep:"magic" }}
```

这个过滤器的定义是：

```
@register.filter
def grep(text, term):
    if text.contains(term):
        return text
```

过滤器的参数总是用引号括起来并用冒号和过滤器名分开。即便你用的是数字或其他非字符串类型，Django的模板语法也要求使用引号。所以假使你写了一个过滤器让它只有在输入文本大于一定长度时才打印的话，它应该是如下这样使用的。

```
{{ bla_bla_bla|hide_if_shorter_than:"100" }}
```

这类标签的实现需要在内部进行一下类型转换，所以这里用到的是int函数。

```
@register.filter
def hide_if_shorter_than(text, min_len):
    if len(text) >= int(min_len):
```

```
    return text
```

基于上述原因，在这里我们显式地把传递进来的字符串min_len参数转换成整数。

虽然有点费解，不过传递给过滤器函数的第一个参数（实际被“过滤”或修改的值）不一定非得是字符串。你看一下日期和时间相关的过滤器就知道此言不虚了，它们都是直接操作Python的datetime对象。所以你完全可以编写一个能处理非字符串对象的过滤器。虽然绝大多数的Web开发都是在和字符串数据打交道，从而让这显得有点不太寻常，不过这的确是可能的。

如果你知道过滤器函数接受了非字符串的数据，但又希望把它们当作字符串来处理的话，你可以给函数加上一个stringvalue装饰器。一个函数可以同时接受多个装饰器，所以如果我们决定把这个装饰器加到 hide_if_shorter_than函数上的话，我们只需在@register.filter下加上一行@stringvalue就行了。

> **注意**
>
> 对函数应用装饰器的顺序是有讲究的。我们刚才提到的顺序（@register.filter，接着是@stringvalue，最后是函数本身）让过滤器接受的输入是一个字符串。要是倒过来把@stringvalue放在@register.filter前面的话，就只能保证过滤器的输出是字符串了。这里的差别虽然很微妙，但是却非常重要。

更复杂的自定义模板标签

你还可以定义更复杂的自定义模板标签——例如，用配对的模块风格（paired block-style）的标签来处理它们所包含的内容。创建这种类型的标签相当复杂，需要直接去操作Django模板引擎的内部机制。这比编写前面那些简单装饰器的工作量要大多了，而且也不是很常用。想要了解更多关于高级自定义模板标签的细节，可以参考Django的在线文档 www.djangoproject.com/documentation/templates_python。

替换模板

Django里模板引擎的工作就是准备字符串来作为HttpResponse对象的内容。当你了解了这一点后，要是基于某些原因你决定不使用Django提供的模板语言，而使用另一种模板引擎也并非难事。

纯文本

以下是一个不使用Django模板机制的最简单的例子。

```
def simple_template_view(request, name):
    template = "Hello, your IP address is %s."
    return HttpResponse(template % request.META['REMOTE_ADDR'])
```

无需任何第三方模块——这个视图只用到了Python内建的字符串模板语法。

选择一种替代的模板机制

Django从一个高度集成的堆叠模型上获得了很多好处。但是它并不是铁板一块——只要你

愿意，绝大多数情况下你都可以把Django的一部分替换成第三方组件。

特别是替换模板系统非常的方便。那为什么需要一个不同的模板系统呢？答案有以下几点：

- 你是从另一个系统转到Django上来的，那个模板的语法用着比较顺手。
- 你还运行了用其他Python Web框架编写的项目，有一个通用的模板语言会比较好。
- 你要把一个项目从其他Python Web框架上移植过来，而且没时间转换现有的模板了。
- 你的表现逻辑需要一些不太容易添加到Django的模板语言里的特性。

使用另一种模板引擎：Mako

在这一节里我们要介绍一个非常流行的第三方模板引擎：Mako。它的样子和设计都和Django的模板语言非常不同，但是它们具有一些相同的优点：速度快，不绑定XML，拥有相似的继承机制。

Mako 取代了Python的模板框架Myghty，而Myghty则又是基于另一个影响深远的Perl系统HTML::Mason。Mako已经被用在了像 reddit.com和python.org这样的网站上，并且同时还是另一个Python Web框架Pylons的默认模板引擎。所以要是你想试试另一种模板系统的话，Mako是一个很不错的起点。它的影响力也波及到了Django的模板系统，所以虽然语法上它们有些许区别，但是概念上两个系统还是有重合之处的。

和Django的模板不同，Mako的语法是基于Python 的。这和Django模板的理念大相径庭，因为后者努力地在其模板里限制编程逻辑的数量。这两种方式各有千秋。Mako的理念是为了尽量方便Python 程序员，而且Python清晰的语法也不会阻挡模板设计师的脚步。

在用Mako编写Django视图之前，以下提供一个简单的例子，看看Mako在解释器中是怎么工作的。

```
>>> from mako.template import Template
>>> t = Template("My favorite ice cream is ${name}")
>>> t.render(name="Herrell's")
"My favorite ice cream is Herrell's"
>>> context = {'name': "Herrell's"}
>>> t.render(**context)
"My favorite ice cream is Herrell's"
```

在第一个例子里，我们显式地传递了变量的名字，而在第二个例子中，我们用的则是熟悉的context。回忆一下，context就是一个传递给render 方法的字典而已。所以这个简单的例子看起来应该也是很眼熟的，因为它和Django模板引擎里的工作方式几乎一模一样。

Mako还拥有和Django很相似的过滤语法。

```
>>> from mako.template import Template
>>> t = Template("My favorite ice cream is ${name | entity}")
>>> t.render(name="Emack & Bolio's")
"My favorite ice cream is Emack & Bolio's"
```

现在我们来把Django视图接驳到Mako模板上去。

```
from mako.template import Template
```

```
def mako_view(request):
    t = Template("Your IP address is ${REMOTE_ADDR}")
    output = t.render(**request.META)
    return HttpResponse(output)
```

这个视图里完成的功能和前面在交互解释器里的是一样的——创建一个新的Mako模板，然后用来自请求的META对象里的context来渲染它，这个META对象我们在第5章里曾经见过。

当然如果你真的打算换用Mako的话，你需要把模板存放在文件系统（或是数据库）上，让Django可以像查找自己的模板那样方便地找到它们（即无须指定完整路径），然后创建一个能理解Mako的render_to_response方法等。所幸这些工作已经有其他Django/Mako的先驱为你做好了。Django Snippets（http://www.djangosnippets.org/snippets/97/）上有一些很有用的代码，页面上还有一些相应的blog帖子链接能指导你使用它们。

11.6 总结

在介绍的时候我们就说到，这一章的内容横跨了多个主题，我们希望让Django程序员开阔一下眼界，并且让你们体会到这个框架真正的灵活性和可扩展性。下一章将会通过介绍另一部分高级话题来给本书的第四部分画上一个完满的句号。

第12章 高级Django部署

和第11章一样，这一章也是由几个相互没有关系的小节组成，分别讨论了不同的主题。第11章讨论的主要还是和应用程序代码有关的话题。而在这一章里，我们要讨论的是一些更加正交的话题，包括部署应用，更新它们运行的环境，以及修改Django本身。

12.1 编写工具脚本

虽然Django是一个Web框架，但是这不代表你就不能在浏览器之外与之交互。事实上，选用Python而不是其他Web专用的语言（比如ColdFusion或PHP）来编写Django的优点之一就在于它可以在命令行环境里使用。比如你要定期或是偶尔对Django应用里的数据进行一些操作，但是却没有要为之创建一整套Web界面的需要。

这些常见的Django工具脚本用例包括：

- 为你的每日构建（或每小时）创建缓存数据或文档
- 把数据导入到Django模型里
- 发送定期的E-mail提醒
- 生成报告
- 执行清理任务（例如，删除过时的会话）

在这方面Python是非常有价值的。你只需要在编写Django工具脚本的Python代码时配上一点Django所需的环境设置就可以了。下面是几个Django工具脚本的例子。我们会逐个解释它们的代码，你可以自行决定哪一个最适合你的项目。

负责打扫清理的Cronjobs

在有大量操作（删除旧记录和添加新记录）的SQLite（和一些PostgreSQL）数据库里，定期清理回收闲置空间是很有用的。拿dpaste.com为例，大多数数据都只在数据库里停留一个月然后就被删除了。这就是说每星期大概有25%的淘汰率。

如果不定期清理的话，数据库很快就会变得臃肿无比。虽然SQLite号称能够支持的数据库文件达到4GB大小，但是我们宁可还是不要测试这个底限。下面就是dpaste.com清理脚本的示例。它每天晚上都会由cron控制执行。（如果是Windows系统，你可以把它设置成"服务"的形式。）

```
import os
import sys
os.environ['DJANGO_SETTINGS_MODULE'] = "dpaste.settings"
from django.conf import settings

def vacuum_db():
```

```
    from django.db import connection
    cursor = connection.cursor()
    cursor.execute("VACUUM")
    connection.close()

if __name__ == "__main__":
    print "Vacuuming database..."
    before = os.stat(settings.DATABASE_NAME).st_size
    print "Size before: %s bytes" % before
    vacuum_db()
    after = os.stat(settings.DATABASE_NAME).st_size
    print "Size after: %s bytes" % after
    print "Reclaimed: %s bytes" % (before - after)
```

在脚本开始的地方，我们在导入两个模块后，手动设置了一下我们的环境——特别是最重要的DJANGO_SETTINGS_MODULE环境变量，这样Django就知道我们要操作的是哪一个项目了。

这个脚本假设Django本身以及你项目的父目录都在Python路径里。它们可以从site-packages符号链接过来，安装为Python eggs，或者包含在PYTHONPATH环境变量里。如果你要在脚本中手动设置它们的话，只要在最初的两个import语句之后加上以下两行代码：

```
sys.path.append('/YOUR/DJANGO/CODEBASE')
sys.path.append('/YOUR/DJANGO/PROJECTS')
```

就可以替换你的路径了——第一行指向的是你系统上Django源码的目录（就像我们这种直接从Subversion里检出运行Django的急进分子）。第二行则是把我们的项目目录加到sys.path里去，这样就可以通过import语句来访问它们了。

记住编写Django工具脚本其实就是编写Python脚本。只要告诉Python知道怎么找到Django和你的项目，以及怎么让Django找到你的设置文件就一切OK了。

数据导入导出

命令行很适合不常运行的工具，但是很不适合最终用户。例如，假设你定期收到数据并且要把它们插入到数据库里去，就可以编写一个Django工具脚本来完成这项任务。

如果你的Django项目构建在SQL数据库之上，你可能会想为什么要绕远去创建一个Python/Django脚本来处理导入呢？直接用SQL不就好了。

答案是通常这些数据都要先经过一些预处理后才能转换为SQL。事实上要是这种导入外来数据格式的工作不是一次性的话，写一个工具来直接处理它们（CSV，XML，JSON，纯文本，或其他任何格式）要比用文本编辑器搜索替换再把数据转换成一大串SQL INSERT语句方便多了。

这里就是功能齐全的（batteries included）Python发挥能量的地方了（特别是它提供了各种能解析大量文本格式的类库）。例如，你要建立一个emai存档，希望导入Unix风格的“mbox”文件的话，就可以选用Python标准库里的email模块，而不必自己动手去写“智能”解析器，它

要么花费掉你大量的精力，要么还错误多多（更惨的是两者皆有）。

下面这个简单的模型可以用来存储email消息——实际上它和purportal.com用来处理垃圾邮件存档的模型非常类似。

```
class Message(models.Model):
    subject = models.CharField(max_length=250)
    date = models.DateField()
    body = models.TextField()
```

假设你手上有了这样一个模块，并且在项目的settings.MAILBOX里也设置好了mbox文件的路径，你就可以像以下这样把邮件导入到模型里去：

```
import os, mailbox, email, datetime
try:
    from email.utils import parsedate  # Python >= 2.5
except ImportError:
    from email.Utils import parsedate  # Python < 2.5

os.environ['DJANGO_SETTINGS_MODULE'] = "YOURPROJECT.settings"
from django.conf import settings
from YOURAPP.models import Message

mbox = open(settings.MAILBOX, 'rb')
for message in mailbox.PortableUnixMailbox(mbox, email.message_from_file):
    date = datetime.datetime(*parsedate(message['date'])[:6])
    msg = Message(
        subject=message['subject'],
        date=date,
        body=message.get_payload(decode=False),
    )
    msg.save()

print "Archive now contains %s messages" % Message.objects.count()
# Depending on your application, you might clear the mbox now:
# open(MAILBOX, "w").write("")
```

这只是一个展示如何编写Django项目脚本的小例子。Python的标准库（假设这里没有第三方的类库）已经相当的完备了。要是你真的打算成为一名合格的Django程序员，你一定要花点时间浏览一下Python的"stdlib"（http://docs.python.org/lib/），至少也要知道它大概提供了什么功能。

12.2　自定义Django codebase

修改Django内部代码是最后不得已才会考虑的手段。这不是因为它很难（Python怎么会难呢，更何况还有那么干净的codebase，以及大量docstring和注释形式的内部文档支持）。我们不鼓励你进入Django内部“修正”问题的原因是很可能根本不需要费那个劲。

Django项目正处于开发活跃的阶段。不过随时和Django的主干或“trunk”版本保持一致基本上是安全的，因为它还是很稳定的。随着新特性的不断添加和旧有bug的不断修正，你可以

紧跟 code.djangoproject.com上的报告并在适合的时候进行升级。但要是你给自己弄了一个山寨版的话，就等于把自己排除在主干的更新之外了。至少你也要花费大量的精力才能把更新和你自己的修改合并起来。如果你有非这么做不可的理由的话，分布式的版本控制系统或许能帮上一点忙（请参见附录C）。

最后，要是你实在忍不住要hack一下Django的话，不妨考虑一下这些为自己所做的修改是不是可以抽象出来让框架的其他用户也受益呢？如果你觉得可行的话，请一定要读一读附录F里向Django项目贡献代码的部分。

12.3 缓存

大流量的网站很少因为Web服务器发送数据的性能而受到限制。这里的瓶颈几乎总是和数据生成有关，数据库响应不够及时，或是服务器的CPU被卡在同一块代码上一遍又一遍地挨个请求执行它们。解决这个问题的方案就是使用缓存——即保存生成数据的一份拷贝，这样就不用每次都去执行相对昂贵的数据库操作或是计算步骤了。

对大流量网站来说，不管后台使用的是什么技术，缓存都是必不可少的技术。Django可以根据网站架构在三种不同层次上为缓存控制提供大量的支持。它还提供了一个很方便的模板标签，允许你特别指明渲染页面里的哪一部分应该被缓存。

基本的缓存方法

Django的缓存框架对初学者来说就像是一堆莫名其妙的配置。虽然每个网站的需求（以及每台服务器的性能）都不一样，不过要是我们用一个实际的例子来开头的话，你就能更好地掌握这个工具。而且，这个配置还正好能够满足大部分网站的需要，所以说不定你在Django缓存里需要关心的就只有那么多了。

画一条底线

缓存的全部意义就在于提升网站的性能，所以在事前做一点评估是很有必要的。每个站点都不一样，所以要知道网站缓存究竟起了多大的作用，只有亲手去测试一下。

ab 是一个基础服务器测试的常用工具，它的全称是Apache Benchmarking tool。如果你已经安装了Apache的话，那么ab也随之一起安装好了。在任何POSIX系的系统上（Linux或者Mac OS X），它应该可以直接访问（已经包含在path路径里了）。在Windows系统上，它可以在Apache的安装目录下找到，例如，C:\Program Files\Apache Software Foundation\Apache2.2\bin（关于如何使用的信息，请参见其手册 http://httpd.apache.org/docs/2.2/programs/ab.html）。

它工作的方式是你提供一个URL和要提交的请求数量，它把最后的性能统计返回给你。这里是一个在第2章里的blog应用示例上运行ab的输出。这里说的直白一点，就是“每秒的请求数”。不要太在意这个例子里的绝对数字，因为它们是运行在一台三年前的笔记本上——你的服务器怎么也比它要快吧！

```
$ ab -n 1000 http://127.0.0.1:8000/blog/
...
Benchmarking 127.0.0.1 (be patient)
```

```
...
Finished 1000 requests
...
Time taken for tests:   27.724 seconds
...
Requests per second:    36.07 [#/sec] (mean)
```

差不多是每秒36个请求。现在我们打开缓存看看有什么不同。

加入中间件

Django的缓存特性是通过一个默认关闭的中间件来实现的。打开settings.py文件，在MIDDLEWARE_CLASSES设置里加上django.middleware.cache.CacheMiddleware。一般你要把它放在最后一行上，因为有些特定的中间件（特别是SessionMiddleware和GZipMiddleware）可能会需要根据缓存框架来修改HTTP的包头。

设置缓存类型

缓存框架提供了至少四种缓存数据存储机制（或后台）。简单起见，这里我们使用的是Django默认的缓存后台，一个叫做locmem的本地内存缓存。它把缓存数据放在内存里，这样存取的速度就非常快。虽然很多缓存方案都选择把缓存放在磁盘上，不过内存型缓存的性能要好的多。（如果你有疑虑，请参见下面关于Memcached的讨论，这是一个原本设计用来支持LiveJournal.com的超高性能缓存系统）。

在settings.py里加入下面一行：

```
CACHE_BACKEND = 'locmem://'
```

（这种独特的伪URL的设置风格在你看过其他类型的后台后就会比较有体会了，它们都使用了这种URL格式来封装配置参数。由于这是默认的后台，严格说起来除非我们要用其他的后台否则不需要设置它。不过Python的专家们说过，“不要依赖默认行为，显式定义你要做的事情”，所以在这里放一个设置的话，以后要修改或添加配置参数都会方便一点。）

实验一下

这就在Django里打开了基本的全站缓存。现在我们来看看带缓存的新网站表现如何。

```
$ ab -n 1000 http://127.0.0.1:8000/blog/
...
Benchmarking 127.0.0.1 (be patient)
...
Finished 1000 requests
...
Time taken for tests:   8.750 seconds
...
Requests per second:    114.29 [#/sec] (mean)
```

settings.py里简单的两行代码就提升了几乎三倍的速度。记住这还是在这个blog应用非常简单的情况下，无论是数据库查询还是商业逻辑都很轻巧，对于更复杂的应用来说，性能的提升将会更显著。

缓存策略

虽然上面这个最简单的缓存实现已经产生了惊艳的成果，但是它们并不能适用所有的情况。我们还没有指定缓存要保持多长时间，缓存不全是网页的内容（例如复杂的边栏或widgets），不需要或不应该缓存的页面（比如admin页面），或是性能调试的参数等等。现在我们就来看看这些话题。

Site-wide

前面我们使用的是全站式的缓存特性。Django会缓存所有没有GET或POST参数的请求结果。最简单的用法我们已经看过了，以下是一些settings.py里的设置它们可以帮助你调试。

- CACHE_MIDDLEWARE_SECONDS：一个被缓存的页面在被一份新的拷贝替换前多少秒内应该被用到。默认值是600秒（十分钟）。
- CACHE_MIDDELWARE_KEY_PREFIX： 用来作为缓存里键的前缀的字符串。如果缓存要在几个Django站点之间共享的话，不管是在内存，文件，还是数据库里，这都能保证在站点范围里不会发生键值冲突。你可以在这些设置里使用任何唯一的字符串——站点的域名或是str(settings.SITE_ID)都是不错的选择。
- CACHE_MIDDLEWARE_ANONYMOUS_ONLY： 简单的基于URL的缓存在交互型的网站里并不是总能发挥作用，因为一个给定URL的内容会随时变化以响应用户的输入。即便你的站点不涉及用户创造的内容 （比如使用Django的admin应用），你也会希望把这个值设置为True，以保证你做的修改（添加，删除，编辑）能立刻反应在admin站点的页面上。

如果Django的页面缓存能符合你的要求，那么上面这些信息基本上就是你全部需要了解的内容了。不过它并不适用于所有情况，我们来看看Django提供的粒度更细的缓存方式及其适用的场景。

Per-view缓存

全站式的缓存假设你站点里的每一个部分都应该被缓存相同的时间。不过你可能有别的打算。比如，你运行的新站点要跟踪每个故事的流行度，然后把这些统计收集起来生成一个最受欢迎的故事列表。显然“昨天最受欢迎的故事”列表可以缓存长达24小时。而“今天最受欢迎的故事”则会不停的发生变化。要在保持内容更新和服务器负载合理之间寻找一个平衡点，我们可能希望这个页面只保持五分钟。

假设这两个列表分别由两个不同的视图生成，那么只要应用一个装饰器就能为它们打开缓存。

```
from django.views.decorators.cache import cache_page

@cache_page(24 * 60 * 60)
def top_stories_yesterday(request):
    # ... retrieve stories and return HttpResponse
```

```
@cache_page(5 * 60)
def top_stories_today(request):
    # ... retrieve stories and return HttpResponse
```

cache_page装饰器只接受一个参数，即这个页面要缓存的秒数。除此之外，你就不用再管别的了。

per-view装饰器利用了所有的Django视图都接受一个HttpRequest对象并返回一个HttpResponse对象的特点，用前一个对象来确定被请求的是哪一个URL，被缓存的数据则用URL为键以字典形式保存起来。然后用后一个对象给HTTP响应设置和缓存相关的包头。

控制缓存相关的包头

讨论缓存到现在，我们主要关注的是怎么设置服务器来决定缓存内容每过多久重新生成一次。在实际使用中，缓存是一个在你的服务器和连接到它上面的客户端之间的对话（包括你可能不能控制的外部缓存服务器）。这个对话由特定的包头控制，即随着HTTP响应传递的“缓存控制”（cache-control）头。

Django给予你的最基本的额外缓存控制形式就是一个“永不缓存”（never cache）装饰器。

```
from django.views.decorators.cache import never_cache

@never_cache
def top_stories_this_second(request):
    # ... we don't want anybody caching this
```

以上代码告诉下层的接收者不要缓存该页面。并且只要它们遵守标准（RFC2616），页面就不会被缓存。这个never_cache装饰器实际是一个对Django提供的更强大灵活的对缓存相关工具的封装：django.views.decorators.cache.cache_control。

cache_control 装饰器通过修改HttpResponse里的Cache-control头来将你的缓存策略告知Web客户端和下游缓存（downstream cache）。你可以设置它的任意六个开关（public，private，no_cache，no_transform，must_revalidate，proxy_revalidate）和两个整数值（max_age，s_maxage）。

例如，要想迫使客户端和下游缓存必须"重新验证"你的页面（检查它是否被改动过，即使它们缓存的内容还未过期），你可以如下装饰你的视图函数：

```
from django.views.decorators.cache import cache_control

@cache_control(must_revalidate=True)
def revalidate_me(request):
    # ...
```

大多数网站都用不到cache_control装饰器这么细致的控制功能。但是在需要的时候，有总比没有好，免得自己动手修改HttpResponse对象的头了。

Django 还允许你控制变化的HTTP头。通常，正文是用URL作为键来缓存的。但是你可以经由一些其他参数来影响一个特定URL返回的内容——例如，已登录的用户可以看到和匿名用户不一样的页面，又或者可以根据用户浏览器的类型或语言设置来自定义响应。所有这些参数

都是通过请求里的HTTP头来通知服务器的。"改变"响应头就能够明确地指明哪些参数会影响到返回的内容。

例如，要根据请求的Accept-Language头，从同一个URL返回不同的内容，你可以告知Django的缓存机制在缓存时要连同包头一起考虑。

```
from django.views.decorators.vary import vary_on_headers

@vary_on_headers("Accept-Language")
def localized_view(request):
    # ...
```

因为“Cookie”头的变化也是很常见的一种情况，所以Django还提供了一个vary_on_cookie装饰器方便你使用。

对象缓存

前面的缓存类型都是专注于缓存页面——若是全站式的缓存，那么就是指网站里的每一页；若是per-view的缓存，那么就是指单独的页面（视图）。这些方案都非常容易实现。不过有时候你还可以利用这个缓存架构来直接保存一块单独的数据。

假设你运营着一个繁忙的网站，其中很多页面都包含了一些需要很复杂的操作才能得到的信息——例如，它可以是一个定期更新的巨大文件的处理结果。相比起来页面的生成倒是很快，既然这个信息要在很多页面上显示，那么在这里使用对象缓存就很有意义了。

Django的对象缓存（实际上就是一组可以设置有效期的键值对）允许你存取任意对象，这样你就可以专心处理那些需要大量资源的对象了。以下的代码来自我们的图片示例，还没有被缓存。

```
def stats_from_log(request, stat_name):
    logfile = file("/var/log/imaginary.log")
    stat_list = [line for line in logfile if line.startswith(stat_name)]
    # ... go on to render a template which  display stat_list
```

虽然第三行里的列表推导式写的很漂亮，但是对于体积巨大的log文件来说它其实也快不到哪里去。我们要做的是避免为每一个请求都去生成stat_list。解决这个问题的主要手段就是django.core.cache里的cache.get和cache.set方法。

```
from django.core.cache import cache

def stats_from_log(request, stat_name):
    stat_list = cache.get(stat_name)
    if stat_list == None:
        logfile = file("/var/log/imaginary.log")
        stat_list = [line for line in logfile if line.startswith(stat_name)]
        cache.set(stat_name, stat_list, 60)
    # ... go on to render a template which  display stat_list
```

cache.get调用会返回任何给定键的缓存值（对象），而如果那个对象过期后，cache.get返回None，并且对象从缓存里被清除出去。

cache.set方法则接受一个键（字符串），一个值（任何Python的pickle模块可以处理的值），和一个可选的有效期（以秒计算）。如果你省略了有效期参数，那么它的值就会设置为CACHE_BACKEND。细节见下。

另外还有一个get_many方法，它接受一个键的列表，并返回一个包含那些键的（还在有效期内的）值的字典。如果你还没注意到，我们在这里告诉你其实对象缓存没有依赖于Django的缓存中间件——我们只需要导入django.core.cache即可，无须修改任何设置或是添加任何中间件。

cache模板标签

Django提供的最后一种缓存机制就是：cahce模板标签。它让你可以从模板里直接使用对象缓存而无须修改视图代码。虽然有些程序员不喜欢这种优化，因为缓存出现在了表现层上，但是也有人觉得它好用。

举例来说，假设有一个模板要显示很长的物品列表信息，而生成这个信息的过程又比较费时费力。同时除了这个列表以外，页面每次载入时都会变化，所以简单地把整个页面缓存下来没什么好处，而且这个长列表最多五分钟才会更新一次。由于这个列表输出的"昂贵之处"是显示循环加上循环里昂贵的方法调用，所以无论是在视图还是在模型代码里都没有一个单独的突破点。不过有了cache模板标签的话，我们就可以直接在需要的地方应用缓存了。

```
{% load cache %}
... Various uncached parts of the page ...
{% cache 300 list_of_stuff %}
    {% for item in really_long_list_of_items %}
        {{ item.do_expensive_rendering_step }}
    {% endfor %}
{% endcache %}
... Other uncached parts ...
```

上面for循环的输出被整个的缓存起来。cache标签接受两个参数：内容要缓存多久（以秒计算），以及缓存里要用的键。

在该例子中，光用静态的键是不够的。假如网站有本地化版本或者要根据当前用户的语言喜好来渲染数据的话，你会希望缓存选用的键能反映出这一点。所幸 cache标签还有第三个专门为这种情况准备的可选参数。这个参数用模板变量的名字加上静态键（在上面的例子里就是list_of_stuff）的名字来创建新的键。

所以为了适应list_of_stuff在每个语言里都不同的这个现实，你的cache标签可以这样定义：

```
{% cache 300 list_of_stuff LANGUAGE_CODE %}
```

> **注意**
>
> 上面这个例子假设你向模板传递了RequestContext对象，它根据你context处理器的设置给模板context添加了额外的变量。默认情况下django.core.context_processors.i18n国际化会被激活并且负责设置LANGUAGE_CODE变量。更多关于context处理器的细节请参见第6章。

缓存后台的类型

前面介绍的Django缓存类型都是“locmem”。以下是一份完整的有关缓存类型的列表：

- dummy：只用于开发，实际上根本没有缓存，不过能让你保持其他缓存设置不变，以便它们能在实际网站上（使用下列后台之一）正常工作。
- locmem：一个可靠的进程安全的内存型缓存。也是默认缓存类型。
- file：一个文件系统缓存。
- db：一个数据库缓存（需要在你的数据库里创建一张特殊的表）。
- memcached：一个高性能，分布式的内存型缓存。也是最强大的缓存类型。

CACHE_BACKEND的设置用的是URL风格的参数，以缓存类型开头，后面跟一个冒号和两个斜杠（如果是文件的话要跟三个斜杠）。开发后台dummy和locmem都不再需要额外的参数。而下面介绍的是file，db和memcached的后台设置。

CACHE_BACKEND设置还接受三个可选参数如下所示：

- max_entries：缓存里保存最多的记录数，默认值是300。记住，通常很小的一部分数据就会占用大部分的服务器资源，所以用不着把所有的内容都放到缓存里才能提升性能。而且由于有效期的工作方式，缓存里应该绝大多数都是最常用的数据。如果你的内存比较小或者缓存里数据过大的话，尝试减小这个值。要是你有很多内存又或者缓存的对象比较小的话，可以尝试增大这个值。
- cull_percentage：这个参数的名字起的很糟糕，它根本就不是代表百分数。它指定了当缓存达到max_entries的时候要删掉多少比例的记录数。默认值为3，即每次缓存满了以后，最老的1/3数据会被删除。
- timeout：数据在缓存里可以存活的时间，默认值为300（秒，即5分钟）。这个数字不单是用来决定哪些数据要从缓存里移除，它还用来创建各种包头来告知Web客户端所发送数据的缓存能力。

这些参数都是以URL参数的风格进行设置，如下所示：

```
CACHE_BACKEND = "locmem://?max_entries=1000&cull_percentage=4&timeout=60"
```

以上代码告诉Django去使用local-memory缓存，最多保存1000条记录，当缓存被填满时移除其中的1/4，并且把缓存数据的有效时间设置为创建后60秒。

文件

文件缓存后台只需要一个Web服务器进程对其有写权限的目录就可以工作了。不要忘了冒号后面要用三个斜杠，前两个是代表URL的组成部分，第三个斜杠则代表这是一个绝对路径（即文件系统的根目录）。和Django里其他文件的设置一样，即使是Windows，这里也是一样用正斜杠。

```
CACHE_BACKEND = "file:///var/cache/django/mysite"
```

当然了，如果是Windows的系统，那么它更可能是这个样子的：

```
CACHE_BACKEND = "file:///C:/py/django/mysite/cache"
```

数据库

你需要先在数据库里设置缓存表后才能使用数据库缓存后台。它的命令是

```
$ python manage.py createcachetable cache
```

最后一个参数是数据表的名字，虽然我们推荐简单地叫它cache就行了，不过你可以选择任何喜欢的名字。设置好数据表以后，CACHE_BACKEND设置就变成了

```
CACHE_BACKEND = "db://cache/"
```

这张表结构很简单，它只有三列：cache_key（表的主键），value（实际缓存的数据）和expires（一个datetime字段，Django在这一列上设置了索引来加快速度）。

Memcached

Memcached 是Django里最强大的缓存机制。同时，它也是设置起来最复杂的一个。但是如果需要的话，这些努力还是值得的。它原本是由 Livejournal.com创建用来缓解他们数据库上两千万pv一天的压力。之后它又被移植到 Wikipedia.org，Fotolog.com，Slashdot.org等其他繁忙的网站上去。Memcached的主页是 http://danga.com/memcached。

Memcached相比其他选择最大的优势就是可以轻易地被分布到多台服务器上。Memcached就是“一张巨大的虚拟的哈希表”。你可以像访问Python字典里的键值对那样使用它，但是它无缝地把你给它的数据发散到很多服务器上去。

虽然Memcached比这里其他的缓存机制要重量级得多，不过它仍然是一个内存型的缓存机制，而不是一个对象数据库。经常有Memcached的FAQ问道，“Memcached有冗余么？”，“Memcached怎么处理故障转移？”，以及“怎么把数据导入导出Memcached？”，答案是“没有，不处理，不需要！”因为你可靠的持久化数据就是你的数据库，Memcached只是让它速度更快而已（关于Memcached创建和架构的详细细节，请参考文章 http://www.linuxjournal.com/article/7451/）。

运行Memcached需要两样东西：其软件本身以及Django用来和Memcached交互的Python绑定。在Linux下你应该可以很容易找到相应的软件包，对于Mac OS X系统则可以去Darwin Ports或Macports查找。Windows下的Memcached可以在这里找到 http://splinedancer.com/memcached-win32。

在运行Django应用的服务器端，你得告诉 Python怎么连接Memcached。纯Python的客户端有python-memcached（http://tummy.com /Community/software/python-memcached）或者速度更快的cmemcache的C库（http: //gijsbert.org/cmemcache/）。python-memcache还可以通过Easy Install来直接简易安装设置。

设置服务器让其在系统启动的时候自动运行memcached守护进程（daemon）。这个守护进程没有配置文件。下面这行告诉memcached以守护进程方式启动，使用2GB内存，监听IP地址为10.0.1.1：

```
$ memcached -d -m 2048 -l 10.0.1.1
```

memcached提供了丰富的命令行选项，如果你有兴趣的话，可以去看看它的手册或是其他

文档。在POSIX系的系统上，把这条命令放在操作系统的启动脚本里就行了，在Windows系统上，你可以把它设置为服务。

有了memcached守护进程之后，你就可以通过CACHE_BACKEND设置告诉Django可以使用它了。

```
CACHE_BACKEND = 'memcached://10.0.1.1:11211'
```

Django会要求指定端口。Memcached默认运行在11211端口上，刚才的命令行里我们没有指定别的端口，所以Memcached服务器就会去监听这个端口。如果有多个服务器的话，可以用分号把它们隔开。

```
CACHE_BACKEND = 'memcached://10.0.1.1:11211;10.0.5.5:11211'
```

最后，虽然Memcached比其他后台需要更多的设置，但是它依然是一个后台，就是说当你正确安装Memcached后，它和其他后台的工作方式是完全一样的。

12.4 测试Django应用

一个公认的观点是拥有一个自动化的测试集对你的应用程序是有好处的。对动态类型的语言来说特别如此，比如Python，它不在编译期为你提供安全的类型检查。

> **注意**
>
> 这一章假设你确实相信测试带来的益处，且专注于如何测试而不是为什么要测试。要是你觉得需要更多的理由来说服你测试，请参考withdjango.com上提供的阅读资料。

Python通过两个补充模块（doctest和unittest）和其他一些流行的独立工具对测试提供了出色的支持。这一章（以及Django）主要关注的是那两个内置的系统，要是你对更广阔Python世界里的测试有兴趣的话，可以参考上面URL里的内容。

坏消息是Web应用很难测试。这是由它们固有的不整洁性造成的，比如各种现实世界的交互，数据库连接，HTTP请求和响应，E-mail生成等。好消息是 Django对测试的支持让你测试项目的时候能稍微轻松一点。在讨论Django的测试之前，我们先来复习一下Python测试的基础。

Doctest基础

一个doctest是一个包含在模块，类或函数的docstring里的Python会话的拷贝。用doctest模块的test runner可以发现这些测试，并能对其进行执行和验证。你可以参阅第1章来复习一下关于docstring的内容和它们的用法。

例如，我们可以轻松地为这个简单的函数编写测试。

```
def double(x):
    return x * 2
```

若是在解释器里手动测试它的话，可以这样：

```
>>> double(2)
4
```

这正是我们要的结果，所以可以宣布函数通过了测试。然后，我们只要把这段交互的会话依葫芦画瓢照搬到函数的docstring就可以给它添加doctest测试了。

```
def double(x):
    """
    >>> double(2)
    4
    """
    return x * 2
```

当用doctest模块的test runner测试这个函数的时候，double(2)会被执行。要是输出为"4"，那么就一切正常。反之则会报错。

test runner还能智能地跳过非测试的文本（比如不是以>>>开头，或是紧跟在它后面的普通文档文本），所以我们还能给函数增加一些可读的介绍。

```
def double(x):
    """
    This function should double the provided number. We hope.
    >>> double(2)
    4
    """
    return x * 2
```

Unittest基础

unittest模块和doctest是殊途同归。它借鉴了Java的测试框架JUnit的理念（JUnit的灵感则是源自Smalltalk中的单元测试框架）。unittest在Python里最典型的用法如下：

```
import unittest

class IntegerArithmeticTestCase(unittest.TestCase):
    def testAdd(self):
        self.assertEquals(1 + 2, 3)
    def testMultiply(self):
        self.assertEquals(5 * 8, 40)

if __name__ == '__main__':
    unittest.main()
```

以上例子是一个完整的脚本。在执行的时候它会执行自己的测试集。这是由unittest.main()调用完成的，它会搜索所有unittest.TestCase的子类，并且调用所有以test开头的方法。

运行测试

Django运行测试的命令是：

```
./manage.py test
```

Django 会自动检测到INSTALLED_APPS设置里列出的所有应用程序里的models.py文件中测试（上面介绍的任意一种）。你还可以给test命令提供额外的参数来缩小测试的范围，指定某个单独的应用，甚至只测试应用里某个模型，例如manage.py test blog或是manage.py test blog.Post。

此外，test命令还会在应用程序子目录下查找任何名为test.py的单元测试文件（和你的models.py同级的目录）。所以你可以随意按照自己的喜好把单元测试放在任意一个地方。

测试模型

模型（model）通常是用doctest来测试的，因为Django会在你运行manage.py test命令时在被安装的应用里寻找它们。如果你的模型只包含数据变量的话，其实没有太多可以测试的东西。因为这时你的模型只是代表了数据表示而已，而Django内部的逻辑早已经过充分测试了。不过一旦你添加了模型方法后，你就引入了需要测试的逻辑。

例如，假设有一个包含了生日变量的Person模型，你有一个根据这个日期计算年龄的模型方法。代码如下：

```
from django.db import models

class Person(models.Model):
    first = models.CharField(max_length=100)
    last = models.CharField(max_length=100)
    birthdate = models.DateField()

    def __unicode__(self):
        return "%s %s" % (self.first, self.last)

    def age_on_date(self, date):
        if date < self.birthdate:
            return 0
        return (date - self.birthdate).days / 365
```

在以上age_on_date方法的代码里，众所周知地容易滋生“fencepost”这类错误，即不当的边界条件（boundary conditions，例如测试一个人的生日）会产生不正确的结果。有了doctest，我们就能防止这类错误发生。

若要手动测试这个年龄方法，可在Python解释器里创建一些样例对象然后调用方法，如下所示：

```
>>> from datetime import date
>>> p = Person(firstname="Jeff", lastname="Forcier", city="Jersey City",
... state="NJ", birthdate=date(1982, 7, 15))
>>> p.age_on_date(date(2008, 8, 10))
26
>>> p.age_on_date(date(1950, 1, 1))
```

```
0
>>> p.age_on_date(p.birthdate)
0
```

以你对doctest的了解，你一定猜到了我们可以直接把这些交互会话的内容放到age_on_date方法的docstring里面去，于是该方法就变成：

```
def age_on_date(self, date):
    """
    Returns integer specifying person's age in years on date given.

    >>> from datetime import date
    >>> p = Person(firstname="Jeff", lastname="Forcier",
    ... city="Jersey City", state="NJ", birthdate=date(1982, 7, 15))
    >>> p.age_on_date(date(2008, 8, 10))
    26
    >>> p.age_on_date(date(1950, 1, 1))
    0
    >>> p.age_on_date(p.birthdate)
    0
    """
    if date < self.birthdate:
        return 0
    return (date - self.birthdate).days / 365
```

最后，运行前面提到的manage.py命令来进行测试：

```
user/opt/code/myproject $ ./manage.py test myapp
Creating test database...
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table myapp_person
Installing index for auth.Permission model
Installing index for auth.Message model
Installing index for admin.LogEntry model
.
----------------------------------------------------------------------
Ran 1 test in 0.003s

OK
Destroying test database...
```

一个小小的测试就产生了这么多输出，但你完整地测试了整个模型层次以后，你会得到一行到两行句号，加上一两个间或出现的E或F（分别代表了意外错误和测试失败）。

最后，虽然doctest能满足你大多数时候的需要，但是如果要测试更复杂的商业逻辑或是模

型关系的时候，也不要对设置模型相关的单元测试有顾虑。要是你是测试新手，可能要花一点时间才能分辨出什么时候要用哪个方法——不过千万不要放弃！

测试整个Web应用

自上而下地测试Web应用不是一桩容易的任务，而且也没有办法用同样的脚本达到100%自动化，因为每个Web应用都是不同的。不过还是有一些工具证明是很有用的。

第一个要关注的就是内建到Django里的工具，在本书编写的时候它还是相当得新。它叫做“Django测试客户端”，你可以在Django官方网站上找到它的文档 http://www.djangoproject.com/documentation/testing/#testing-tools。测试客户端提供了一个简单的方法来模仿请求－响应循环，并测试特定条件。

如果你需要比内置测试客户端更多的控制的话，可以试试更成熟，特性更丰富的工具Twill（http://twill.idyll.org/）。和Django的测试客户端一样，它完全基于命令行，易用却不失强大——典型的Pythonic类库。

另一个最近比较火的测试工具是Selenium（参见 http://selenium.openqa.org/）。和其他两个不同，这是一个基于HTML/JavaScript的测试工具，专门从浏览器的角度来测试Web应用。它支持绝大多数平台上的主流浏览器，而且因为它是基于JavaScript的工具，所以还可以测试Ajax功能。它能分成2.5到3 种操作模式：Selenium Core，Selenium RC（Remote Control，远程控制）和Selenium IDE（Integrated Development Environment，集成开发环境）。

Selenium Core（http://selenium-core.openqa.org/）是（手动和自动）测试Web应用的核心。有人将它称为“机器人模式”（bot mode）的Selenium。粗活重活都由它干了。除了Web应用的系统功能测试外，它还可以进行浏览器兼容性测试。

Selenium RC（http://selenium-rc.openqa.org/）让用户可以用不同的编程语言创建完整的自动化测试。你自己编写的测试应用，Selenium Core会运行它们——你可以把它想象成在Core之上的一个脚本层（scripting layer），或者“编码模式”（code mode）。

使用Selenium最好的起点就是它的IDE（http://selenium-ide.openqa.org/）。这是一个Firefox扩展同时也是一个完整的IDE，你可以将Web会话记录下来然后通过回放来测试。它还能将测试输出为各种Selenium RC支持的语言，以便你进一步增强或修改它们。你可以设置断点然后一步步调试这些测试。因为它是写在Firefox上的，所以一个经常会被问到的问题就是有没有Internet Explorer（IE）版。答案是没有。不过IDE的“记录模式”（record mode）可以让你通过Selenium Core来运行它们。

除了上述三个工具（Django测试客户端，Twill和Selenium）之外，你还可以在 http://www.awaretek.com/tutorials.html#test 以及里面的链接上找到更多关于Web应用测试的内容。

测试Django代码本身

Django框架自身包含了大量的测试集。每一个bugfix都会配合有一个回归测试来保证已修正的bug不会悄悄重现。新的功能也都配有完整的测试来保证它按照预期正常工作。

你可以自己运行这些测试。当你在一个比较少见的平台或是一个不常见的配置下运行Django有问题时非常有用。虽然你总是应该先检查自己的代码，不过也有可能是你发现了一个之前从未见过的不寻常bug。在内置测试集上的失败会为你生成一份bug报告，以便稍后仔细检查。

运行Django的测试集相当简单，只需要一点小设置即可：你需要指向一个配置文件，这样它才知道怎么创建测试数据库。这可以是任何项目的配置文件，又或者你可以创建一个空项目（即没有应用的项目），然后只在settings.py文件里填入DATABASE_*的设置。

在Django安装目录顶层有一个叫tests的目录（不要和整个Django软件包里的test包混淆了），其中就包含test runner其调用命令如下：

```
$ tests/runtests.py  -settings=mydummyproject.settings
```

这是一个相对安静的过程，因为测试只有在失败的时候才会生成输出。由于测试集的运行需要一段时间，所以你会在过程中不断收到反馈。runtests.py命令接受一个-v verbosity的参数。在-v1下的输出是这样的：

```
.....................................................E...EE...
```

以上E字符代表了一些测试出现了错误，后面会显示出具体失败的总结，以便你界定这是人为的设置失误还是Django确实有问题。

详细程度更高的-v2输出以一长串import开始，然后是创建测试数据库和数据表的细节信息（这里的“...”代表为节约篇幅，剩下的输出都省略了）。

```
Importing model basic
Importing model choices
Importing model custom_columns
Importing model custom_managers
Importing model custom_methods
Importing model custom_pk
Importing model empty
...
Creating test database...
Processing contenttypes.ContentType model
Creating table django_content_type
Processing auth.Message model
Creating table auth_message
Processing auth.Group model
Creating table auth_group
Processing auth.User model
Creating table auth_user
...
```

在运行测试集过程中看到一个失败不一定代表你发现了Django的错误——如果你不确定的话，最好先到Django的用户邮件列表上把你的配置细节和测试失败的输出贴出来给大家讨论一下。

12.5 总结

这一章覆盖了好几个高级的主题，算上第11章，我们希望你能充分了解到Django开发的深度。当然，这些主题只是一个展示可能性的示例：Web应用开发和其他大多数计算机分支一样，并不是一个自我封闭的系统，它还涉及了很多更广泛的方方面面，就好像Python一样，能够处理各种情况和技术。

到此，你几乎就要读完这本书了——恭喜！剩下的还有附录，和最后两章一样，它们也是本书重要的一部分，涵盖了很多不同的主题，从命令行使用到安装部署Django，还包含一系列的外部资源和开发工具等。

最后，你可能还会想要回过去重新读一下（至少是浏览）本书前面的部分。在了解了我们所有要讨论的东西后，之前的代码示例和解释或许能让你温故知新。对任何技术书籍来说都是如此，本书也不例外。

附 录

附录A 命令行基础
附录B 安装运行Django
附录C 实用Django开发工具
附录D 发现，评估，使用Django应用程序
附录E 在Google App Engine上使用Django
附录F 参与Django项目

附录A 命令行基础

绝大多数的Web服务器（更不要说E-mail服务器、文件服务器等）主要都是运行在遵循POSIX的操作系统上，比如 Linux、FreeBSD，还有各种类UNIX系统等，Django的Web服务器当然也不例外。几乎所有的Django核心团队以及很大一部分的社区都是在这样的机器上运行框架。如果你之前从来没有接触过在这种环境中常见的命令行界面的话，本附录会为你做一个基本的介绍，这能让你更加容易理解本书中的例子。

如果你来自Windows的世界，本附录里的知识可能不是那么直观，不过我们还是建议你阅读一下（至少浏览一遍吧），将来总会用到这些东西的。而且通常来说，一个程序员接触的语言、平台和技术越多，他越是能够更好地利用这些现有的和新的工具。

要是你希望在Windows系统上尝试这些命令的话，可以试试一个模仿Linux的环境Cygwin。它由一个模拟层以及一系列UNIX用户常用的命令行工具组成，在这一节里将会介绍其中的一些。但是它可不能就此把你的PC机变成一台服务器。更多关于Cygwin的信息可以从http://cygwin.com获取。

如果你是Mac的用户，那算你运气。Mac OS X是从BSD（Berkeley Software Distribution）Unix继承而来的一个分支，所以它具有一台服务器的许多功能。你只需打开终端（Terminal）应用程序（可以在 /Application/Utilities下找到），就可以输入命令了。从现在开始，我们假设你已经进入UNIX的命令行模式下了。

A.1 在命令行模式下输入命令

和点击鼠标、填写文本、用鼠标驱动的界面不同，类Unix的服务器操作系统是由命令行解

释器或者说shell来驱动的，文字提示符接受命令并且逐个执行。如果你有编程的经验，你一定熟悉编程语言的表达式：打印一个字符串、向调用的函数传递一些参数等。命令行基本上也差不多。

下面这个简单的例子展示了如何列出我们当前目录（在Windows里叫“文件夹”）的内容，列出一个子目录的内容，以及从子目录里删除一个文件。注意$是一个提示符，在下面的例子里用来区分哪些是输入的命令，哪些是命令的输出。

除了$，每个shell都可能使用不同的字符来作为提示符，比如>或者%。(Python解释器用的是>>>提示符。）除了这个出现在用户输入前的符号之外，很多提示符还可以给出更多的信息，例如当前的用户名、机器名或者当前目录等（拿本书的某些例子来说，就是这样的形式：user@example $)。

下面是一个简单的例子，它列出了当前目录的内容，并且从子目录里删除一个文件：

```
$ ls
documents  code  temp
$ ls documents
test.py
$ rm documents/test.py
$
```

这里用到的两个命令都是位于当前执行路径中的程序，或者说二进制文件（下一节会详细讨论路径)。虽然理论上程序的行为是没有限制的，但是我们还是设立了一些指定参数和选项的标准方法。Unix命令一般由三个部分组成：命令的名字，控制命令行为的选项，指定运行什么子命令和所要操作的文件等等的参数。

拿上面这个例子的最后一条命令来说，rm是程序的名字(rm就是remove，代表删除的意思)，documents/test.py是一个参数，指明了要删除的文件。如果要删掉整个目录，可以向rm传递一些选项来控制其行为，如下所示：

```
$ ls temp
tempfile1  tempfile2
$ rm temp
rm: cannot remove 'temp': Is a directory
$ rm -help
Usage: /bin/rm [OPTION]... FILE...
Remove (unlink) the FILE(s).

  -d, -directory       unlink FILE, even if it is a non-empty directory
  -f, -force           ignore nonexistent files, never prompt
  -i, -interactive     prompt before any removal
      -no-preserve-root do not treat '/' specially (the default)
      -preserve-root   fail to operate recursively on '/'
      -r, -R, --recursive   remove the contents of directories recursively
      -v, --verbose         explain what is being done
          --help     display this help and exit
          --version  output version information and exit
```

```
$ rm -rf temp
$
```

在一般情况下，rm是不能删除目录的，这就是为什么一开始我们失败了。但是传入-r和-f选项后——方便起见，你可以只用一个选项符号把它们组合在一起（详见稍后的A.2节）——就可以让rm强制递归地删除目录了，所以它就可以无障碍的删除temp目录。

在上面这个例子里我们看到，程序通常都内置有帮助信息来解释选项和参数的作用。UNIX系统上几乎所有的程序都有-h或--help选项，它会打印出关于程序的帮助信息。不管是新人还是专家，这些信息通常都能有足够的帮助。

注意

你的输出可能和这里有所不同，这是因为UNIX系统的变形实在是太多了，每个平台上对这些程序的实现都会稍有不同。即便是rm和ls这样的核心命令也不例外，不过它们之间至少有一部分是一样的。

如果内置的帮助也不能满足你，或者你想要关于某个选项更详细信息的话，可以参考man系统（就是用户手册manual的意思），它提供了每一个命令的完整信息，更加详尽地解释参数的作用，有时还会给出例子。当然，man自身也是一个程序，一般它只接受一个参数，即你希望查阅文档 man page的程序名。一个经常给UNIX新手举的例子就是man man，即man命令自己的文档手册。

先不说好坏，类UNIX系统通常是面向自学者的。既然你不讨厌阅读，那么作者也乐意相信你会学好的。而且说真的，养成这样一种习惯对你自己也是非常好的一件事情：老鸟们对那些懒得自己寻找答案的菜鸟批判起来通常都是毫不客气的。

UNIX程序的命名

很多UNIX程序的名字看上去都非常莫名其妙，像什么rm、ls、sed等。这些名字有些是因为历史原因（今天我们习以为常的键盘和显示器曾经都是速度很慢的），有些是因为输入起来比较方便。在一个绝大多数是键盘输入的环境下，显然少花点时间在打字上相应的就能更快地完成工作。

有些简写的名字还是比较容易辨别的，比如rm（移除remove）和ls（列出list），而有些就晦涩一点，比如sed代表的是“stream editor”（流编辑器）。其他的一些，特别是较新的程序的名字，则是通过各种不同的缩写组合起来的，例如流行的编译器gcc代表的是“GNU C compiler”，这里的GNU指的是GNU项目，它本身也是“GNU's Not UNIX”的缩写。

总的来说，UNIX命令都喜欢用缩写来表示——一旦你知道了这些命令的作用后，你自然而然的就会理解这些简写名字的涵义了。

A.2 选项和参数

前面我们说到选项和参数是程序规范里两个完全不同的部分，其实这么说不完全正确。在

程序内部，选项和参数只不过就是一个传递给程序的长字符串而已，程序不管按什么规则解释它都可以。因此，这里的标准非常简单：完全取决于程序的作者在多大程度上严格遵守了标准。

通常程序所有的参数和选项都是用空格隔开，选项一般出现在参数之前，以连字符-作为前缀开始，而参数则没有前缀。有的程序还接受一种叫“长选项”的格式，通常这种格式使用两个连字符再加上选项的完整名字——比如前面提到的--help选项。

```
$ rm --help
Usage: rm [OPTION]... FILE...
```

标准是这样写的：工具名/程序名，后面跟零个或者多个任意类型的选项，再跟零个或者多个参数（有的程序根本不接受参数）。选项可以是一次一个，用空格分开：

```
$ rm -r -f temp
```

也可以把这些选项组合起来，少打几个字符，就像我们之前做的那样：

```
$ rm -rf temp
```

不过请注意你不能把短选项和长选项组合在一起，因为那样做是没有意义的，但是下面这样是可以的：

```
$ rm -rf --verbose temp
```

根据选项的不同，它们自身也可以携带参数。例如，head程序的作用是返回给定文件或者文本的开头几行。下面的例子展示了返回的行数是可以通过-n参数来控制的，这里返回了myfile.txt文件的前五行：

```
$ head -n 5 myfile.txt
```

在默认情况下（即不带-n选项），head会显示头十行内容。另外，就算在多个选项组合到一起的情况下，选项的参数也不一定要和它的选项用空格隔开，即放在一起也是可以的。

```
$ head -n5 myfile.txt
```

最后，相比传统的UNIX程序基本上都是严格的按照<program> <options> <arguments>的顺序解析命令来说，很多Linux的应用更加宽容一点，例如：

```
$ head myfile.txt -n5
```

或者：

```
$ rm -rf temp --verbose
```

虽然Linux程序的这种风格非常好用（比如你直到命令打完了才想起来忘了某个选项的时候），我们还是建议你尽量地遵循正统UNIX系统上那种更严谨的格式。否则到时候你会发现在使用FreeBSD或是Mac OS X时，不停收到程序警告你的参数顺序。相信我，我们吃过这样的苦头。

这里的例子都为了展示概念而尽量的保持简洁，要是你在网上查找这些命令行程序用法的例子（或是查阅各种man page和--help输出）时，你会发现这些命令行程序提供了大量的功能和灵活性来改变它们的行为。在下一节里，我们会介绍一种完全不同的UNIX命令行工作方式。

A.3 管道和重定向

本质上来说，命令行主要是处理文本的输入输出。但是除了从用户那里获得输入输出之外，UNIX程序还可以通过一种叫管道（pipe）的I/O抽象机制来和磁盘上的文件交流。顾名思义，管道是一种能在终端用户、程序和文件之间定向文本流的机制。

每个UNIX程序都能处理三种潜在的输入输出：输入、常规输出和与错误有关的输出。在一般情况下，程序交互的是所谓的“标准”管道，即用户面对的文字终端。例如，当你使用cat命令（就是连接concatenate）来导出文件的内容时，cat会打开参数中所列出的文件并且把它们的内容放到标准输出流（stdout stream）里。就跟在下面这个例子里，我们cat一下grocery列表的内容一样。

```
$ cat groceries.txt
Milk
Canned corn
Peanut butter
Can of soup
Powdered milk
```

这里直接运行cat的结果是stdout会把文件的结果输出到终端上。要是cat发生错误，比如给定的文件不存在的话，它会向标准错误流（stderr stream）输出一个错误消息，在默认情况下也是直接返回给用户。

```
$ cat foo.txt
cat: foo.txt: No such file or directory
```

管道抽象的巧妙之处在于当使用管道操作符 | 的时候，它告诉命令将一个程序的标准输出重定向给另一个程序的标准输入。程序I/O的第三种类型就是stdin，标准输入（standard input）。很多程序都能额外地或者只从stdin接受文本，而不仅仅是要求用户提供文件的名字。

举例来说，我们用grocery列表来重新看一下head的用法，这次我们用它来查找列表中的第一项是什么。

```
$ cat groceries.txt | head -n1
Milk
```

注意这里我们没有告知head要处理的文件，而是用管道将cat命令的输出重定向给了head命令。其结果和我们直接将文件传递给head是一样的。

```
$ head -n1 groceries.txt
Milk
```

一个更真实的例子是使用grep工具，它可以用来返回匹配给定正则表达式（参见第一章）的那些行。我们来对grep命令的输出用一下管道看看，这里grep的作用是（以大小写不敏感的方式）把我们grocery列表中不含“can”这个词的行都过滤掉：

```
$ grep -i "can" groceries.txt
Canned corn
Can of soup
```

然后用head来把结果限制在第一项上。

```
$ grep -i "can" groceries.txt | head -n1
Canned corn
```

前面说过，我们可以在一个命令里使用多个管道。现在让我们来用grep的姐妹命令sed把单词“corn”替换为更通用的“veggies”。

```
$ grep -i "can" groceries.txt | head -n1 | sed -e "s/corn/veggies/"
Canned veggies
```

我们说过你可以把文字流用>和<字符重定向给文件。和管道从左到右的作用方式一样，>用来将stdout重定向给文件，而< 则是将文件重定向给stdin。例如在上面的例子中我们完成了查找和替换，虽然它不是那么有用，但是一旦结果显示到屏幕上以后，我们的劳动成果就丢失了（当然，复制粘贴除外）。现在让我们来把它重定向给一个新文件。

```
$ grep -i "can" groceries.txt | head -n1 | sed -e "s/corn/veggies/" > filtered.txt
```

这条命令创建了一个叫做filtered.txt的新文件（小心，它也能覆盖现有文件！），其内容为“Canned veggies”。注意这条命令不会在终端上产生任何输出——因为我们已经把stdout重定向给文件了，所以当然看不到啦。

最后，你还可以用两个重定向符号（>>）来将输出添加到现有文件的末尾，而不是覆盖它。和>一样，如果新文件不存在的话，它会自动创建。

A.4 环境变量

命令行shell里有一种叫环境（environment）或者是名字空间（namespace）的东西，和Python等其他语言里的namespace 一样，它可以保存很多绑定了变量名的不同的字符串，用户在执行命令时可以引用它们，甚至命令自身也可以使用它们（即访问用户的environment）。执行env命令会打印出当前状态下的environment，如下所示：

```
$ env
TERM=linux
SHELL=/bin/bash
USER=user
PATH=/usr/local/bin:/usr/bin:/bin:/usr/games:~/bin
PWD=/home/user
EDITOR=vim
HOME=/home/user
```

很多环境变量都是给常用的UNIX shell工具或是shell本身使用的——比如Subversion就会用到EDITOR变量调用指定的程序来暂时保存用户的输入。TERM变量则决定了终端的类型，很多程序在决定是否使用着色输出或是如何解释击键的时候都要引用它。PWD则是当前的目录等。

和Python以及其他语言一样，你可以用赋值运算符来给这些变量赋值。简单起见，这里只列出了env的一小部分输出：

```
$ env
EDITOR=vim
$ EDITOR=pico
$ env
EDITOR=pico
```

在该例子中，默认的EDITOR值是著名的vim编辑器，我们让它指向了一个不那么强大的编辑器pico。但是和你猜想的一样，环境变量是在每个shell会话开始的时候初始化的。shell启动的时候都会去读shell的配置文件，除非我们对这个配置文件作出修改来永久的保留设置，否则我们对 EDITOR所作的改变只是暂时的（详细内容请参考你shell的man page）。

我们已经看到了env可以打印环境变量的值，但是和Python的全局函数一样，除非你在排错调试，否则这个不是什么有用的功能。shell在碰到以$字符开头的环境变量时会自动查找并替换它的值。这里我们用echo程序来展示这一点，echo的功能非常简单，就是把它的参数返回给用户：

```
$ env
EDITOR=vim
$ echo EDITOR
EDITOR
$ echo $EDITOR
vim
```

可以看到，echo EDITOR自身时shell不会进行任何特殊操作（EDITOR只是一个字符串而已），但是echo $EDITOR时shell就会打印EDITOR变量的值。换言之，shell会在碰到$开头的名字和表达式时尝试把它展开为变量的值。如果你熟悉那些用$字符表示变量的语言的话，一般都犯过类似这样的错误：

```
$ $EDITOR=pico
-bash: vim=pico: command not found
$ EDITOR=pico
$ echo $EDITOR
pico
```

最后要说的一点是：环境变量不是只能保存单个单词的简单字符串而已，它能保存任何字符串。在上面的例子中，shell展开$EDITOR后，将它和行里剩下的内容拼在一起，然后试图将它当作一个命令来执行。这显然是行不通的（因为没有叫vim=pico这个名字的程序），但是我们完全能够利用这一点来偷懒少打几个字符，比如下面的例子，我们将一个命令字符串赋值给变量，然后加上不同的参数重复使用。

```
$ FINDMILK="grep -ni milk"
$ $FINDMILK groceries.txt
1:Milk
5:Powdered milk
$ $FINDMILK todo.txt
1:Search grocery list for milk
$ $FINDMILK email_from_reader.txt
3:Also, what's up with all the groceries and milk examples?
```

我们的FINDMILK变量每次都被shell展开，变成grep -ni milk groceries.txt这样的命令。不过这个例子有点太矫情了——在大多数这种情况下，你真正需要的是参数化这个静态的程序调用，写一个简单的 shell脚本就好。例如你写一个接受两个参数的脚本，用来指定要查找的字符串以及在哪里查找。

shell脚本编程的细节已经超出了本章的范畴，你可以参阅有关shell详尽的man page，网上也有大量优秀资料可以查阅。

A.5 路径

可能最重要的环境变量之一就是路径了，通常都是保存在PATH里，它包含了一系列目录，当你键入命令时shell会在这些目录里查找。

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/games:~/bin
```

当你在shell里输入一条命令时，shell会逐个检查每一个列出的目录直到它找到你要求的可执行文件，然后，它就会用你提供的参数和选项运行那个文件。比如，当我们键入echo时，shell真正执行的是/bin/echo，当键入man时，它找到并执行的是/usr/bin/man。（下面会解释这些不同的bin目录。）

> **Unix执行文件目录**
>
> Unix系统传统上为不同类别的程序准备了多个目录，比如/usr/bin存放的是普通用户的二进制文件（即可执行文件，就和那些我们已经看到的工具一样），而/usr/local/bin则是存放用户安装的程序（即不是随操作系统安装的程序）。而其他的二进制文件，诸如属于系统管理员而非普通用户的程序，则是存放在/sbin和/usr/sbin之下。
>
> 有些第三方应用程序，比如Java实现或旧版的Mozilla套件，会把它们的数据放在/opt/media下，而把执行文件放在/opt/Mozilla/bin下面。路径通常包含了系统执行文件目录的一部分，第三方执行文件目录和其他目录等。

你可以在路径里添加内容来节省打字的时间。在前面的例子里，用户把~/bin加到了路径里，因为~代表的是用户的home目录，这意味着当用户直接输入他们个人执行文件目录里脚本的名字时，shell可以轻松地找到它们。

当然对不在路径中的可执行程序也可以通过指定完整或相对的路径来执行它们。事实上，shell提供的路径功能相当的方便和快捷。

```
$ /tmp/packagename-2.0.1/bin/program
```

由于/tmp/packagename-2.0.1/bin/（如果你的程序没有完整安装的话，这可能是可执行文件的位置）不太可能在你的PATH路径里，你必须明确的告诉shell怎么找到它。虽然非常简单，不过路径这个概念为你提供了一条捷径。

最后记住，在PATH里包含的应该是包含执行文件的那个目录，而不是执行文件本身——

这一点和Python的模块路径是一样的道理（参见第1章）。把路径想象成一组容器，而不是一组要引用的东西。

A.6 总结

到此你已经学习了相当多的内容：如何用不同的选项和参数运行程序，通过管道和重定向来让这些程序一起和文件工作，以及怎样用环境变量和路径设置来节省大量的时间。

但是我们在这里讲授的东西离即使是普通的UNIX shell程序也还远着呢——很多shell自身就是一个完整的编程环境，包含有条件语句、循环等等。当熟练使用文件系统和运行命令后，你会发现花点时间深入了解一下系统的shell是非常有好处的，和其他编程工具一样，它能为你节约大量的时间和精力。

附录B 安装运行Django

开发Django首先当然要进行安装，而安装的方式则取决于你选择的操作系统和手头上的工具。运行Django最简单的环境包括Python 2.5，一个轻型的SQL数据库包，以及Django内置的开发Web服务器。

不过一般的Django开发都会构建在一组更稳定的应用程序上，比如工业强度的Apache、PostgreSQL等。这一节里，我们会向你介绍各种Django开发的选项，包括一些最常见的配置等，并将其分成几个核心部分逐个讨论，即Python和Django本身，Web服务器，和数据库引擎。

B.1 Python

和大多数语言一样，版本总是新胜于旧。Django可以运行在任何Python 2.3以上的版本里，不过我们推荐还是用你手头上最新最稳定的版本比较好。（在编写本书时，Python正在从2.5.2向2.6过渡，同时还有下一代的新版本3.0。）请参考第一章来更多地了解有关这些版本的差异。

在Python的官方网站http://www.python.org/download/上可以下载到大多数主流平台的安装包。下面我们简要介绍一下不同平台上要注意的地方。

如果不确定你机器上安装的Python是什么版本，可以打开命令行终端并且键入python-V查看（注意是大写的“V”）。如果Python正确安装的话，它会显示版本号然后退出。

```
$ python -V
Python 2.5.1
```

Mac OS X

Python 在Mac上是默认安装的，但是除非你使用的是自带Python 2.5的OS X 10.5（“Leopard”）的话，你Python的版本应该是2.3，我们推荐你最好升级。Python 2.4和2.5都可以直接从python.org获得安装包，或者你也可以通过MacPorts（http://macports.org）这样的软件升级系统来安装。如果你决定使用MacPorts的话，你还需要安装python_select包，这样你就可以把新版本设置为系统默认版本。

Unix/Linux

大多数开源的类Unix系统如Linux和BSD家族等也都将Python作为核心系统的一部分默认安装了。具体的版本取决于你的系统和更新的方式。检查你发行版的包管理系统以确认你安装了最新版的Python，或者你也可以从python.org的下载页面上获取单独的安装包，比如RPM或是直接下载源码。

Windows

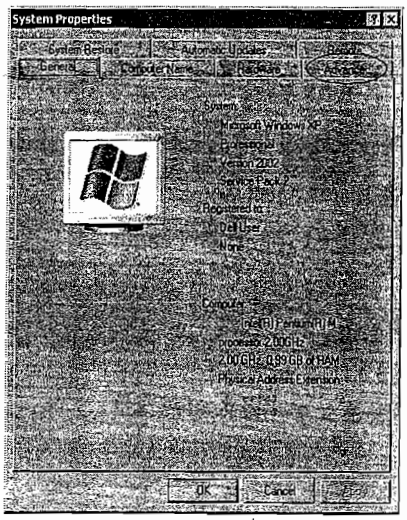
Windows系统默认不自带Python，所以你需要到官方网站上去下载。或者你可以访问《Core Python》一书的网站http://corepython.com上的Python下载页面来获得各种平台上Python的最新版本。

另外，还有一个可选的Windows平台专用的Python库，叫做Python Extensions for Windows（即win32all）。这个库让你能直接开发原生的Windows Python应用。就算Python的新用户还不需要这种系统集成，他也会发现PythonWin这样的IDE需要这个包。

更新你的路径

Python 安装完成后，你可能还需要把可执行文件添加到系统的路径里去。在Linux和Mac OS X这样的类Unix系统上，如果Python是安装在常见的/usr/bin，/usr/local/bin等目录下的话，这一步通常都已经自动完成了。但是Windows用户则需要手动地把它添加到路径里去，步骤如下：

右键点击“我的电脑”，选择“属性”进入系统属性对话框。然后选择“高级”标签，如图B.1所示。

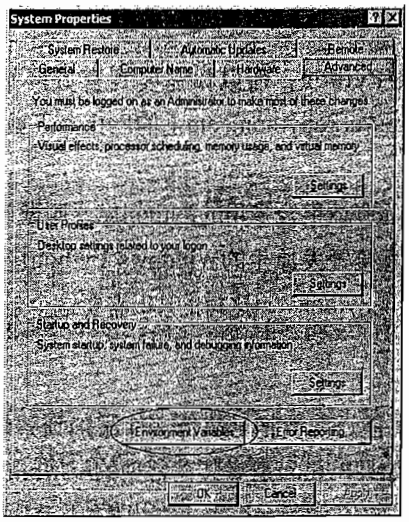


图B.1 系统属性

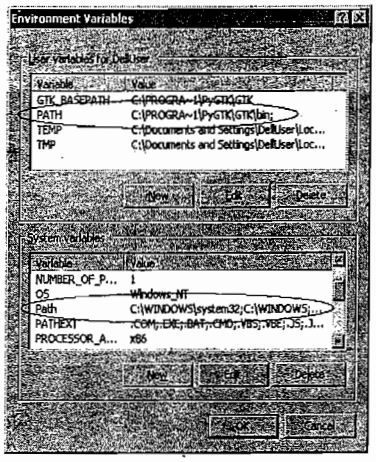
你会看到三个主要的部分（如图B.2）。跳过这些直接点击窗口下方的“环境变量”按钮。

然后，你会看到有两个面板（如图B.3）可以让你修改环境变量。你可以选择只是添加/修改/更新你自己的路径变量（“USER的用户变量”）PATH，还是为整个系统（“系统变量”）作出修改。这还取决于你是否有权限那么做。

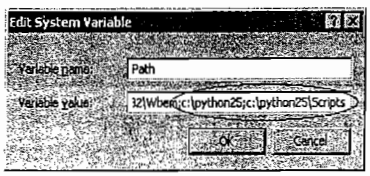
选择你要修改的变量然后点击“编辑”按钮，按照格式把C:\Python25添加到列表里，如图B.4所示。



图B.2 高级页



图B.3 环境变量



图B.4 修改路径

如果PATH里已经有文件夹了，你可以把它添加到任何位置，只要保证所有的文件夹都是用分号隔开的就行了。若还没有这个变量，那直接添加就可以了——只有一个文件夹是完全正确的。

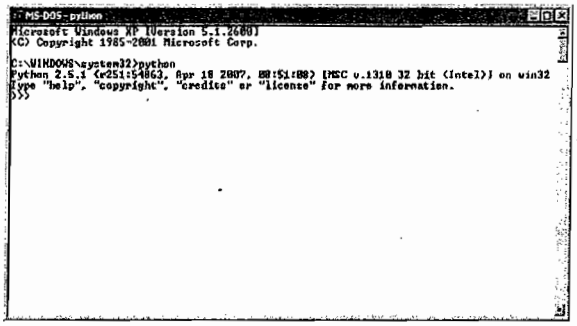
当你点击OK后，可以打开一个新的DOS命令行窗口，这时你应该已经可以不需要完整的路径C:\Python25\python.exe，而只需键入python就能启动解释器了（参看下一节）。

测试

要确认你的Python安装正确，只需通过运行Python主程序启动交互解释器就可以了。在shell或是终端窗口里输入python（可能还需要带上版本号，比如python2.4），如果一切正常的话，应该可以看到如下的显示结果：

```
$ python
Python 2.5.1 (r251:54863, Mar  7 2008, 04:10:12)
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

如果是Windows上的DOS窗口的话，应该如图B.5所示。



图B.5 cmd.exe下的Python

三个大于号提示符（>>>）表示你正在交互解释器的提示符下，你可以键入任何合法的Python代码。要退出可以输入Ctrl-D（UNIX shell或是IDLE）或者^Z（DOS命令窗口）。

如果你得到一个类似"command not found"（找不到命令）或是"'python' is not recognized as an internal or external command"（'python'不是一个可识别的内部或外部命令）这样的错误信息的话，说明你没有把文件夹正确地添加到PATH变量里去，请重新检查你的设置，仔细检查你的安装路径，确认python.exe的存在后，再把文件夹的名字加入到PATH里去。

最后为方便起见，你还可以把Scripts文件夹按照上面的步骤也添加到路径里去。这样你就可以用和启动Python解释器一样的方法使用Django的管理工具django-admin.py了。

如果这是你第一次使用Python，而且你还没读过任何教程或是本书的第一章的话，我们建议在继续下去之前至少先粗略的看一下它们。在正式开始使用任何新工具之前，稍微做点测试总是有好处的。

可选的插件

除了Python，这里还有两个值得推荐的（但不是必须的）工具：Easy Install和IPython。

Easy Install

Python 最强大的特性之一就是它默认提供了一个丰富的标准库来帮助你完成任务，相当的“傻瓜”。要是这还不够的话，还有更多的第三方库可供选择。所以在你重新工作之前，先到

Python Package Index（或者叫“PyPI”，http://pypi.python.org）上瞧瞧有没有适合你的工具。

如果知道了你需要的工具，管理你的Python安装就变得麻烦起来。你得关心Python版本的兼容性，软件之间的依赖关系，以及将新的模块包集成进来以便你的应用程序可以导入它们等。

好在有一个叫Easy Install的工具可以帮你管理这一切，它的位置在http://peak.telecommunity.com/DevCenter /EasyInstall。你只需下载一个ez_setup.py文件，运行它就可以了（需要sudo或者管理员权限）。它能大大简化新软件包的安装过程，如下：

```
easy_install NEW_3RD_PARTY_SOFTWARE
```

Easy Install通过“PyPI”来获取所需软件的最新（或者你所要求的）版本，自动下载（及其依赖的软件包）并且为你安装，这一切只需要一个简单的shell命令就可以完成了。

同样，升级和卸载软件包也是一样的简单。

IPython

IPython 是一个随Python发行的标准交互解释器的第三方实现。它在Python原版的基础上添加了许多有用的功能，包括自动完成变量名和属性名、命令行历史、自动对齐、可以轻松的访问docstring和参数签名等。基于这些优点，当你用Django项目的manage.py管理脚本启动shell 时，它会将其作为默认的解释器。更详细的关于IPython的信息可在http://ipython.scipy.org下找到。

作为从网站下载以外的另一种方式，同时也是为了介绍Easy Install和IPython（特别是展示用Easy Install安装第三方软件包的便捷性），这里以通过Easy Install在Linux系统上安装IPython为例试举一例（用sudo来获得超级用户权限）：

```
$ sudo easy_install ipython
Password:
Searching for ipython
Reading http://pypi.python.org/simple/ipython/
Reading http://ipython.scipy.org
Reading http://ipython.scipy.org/dist
Best match: ipython 0.8.4
Downloading http://ipython.scipy.org/dist/ipython-0.8.4-py2.4.egg
Processing ipython-0.8.4-py2.4.egg
creating /usr/lib/python2.4/site-packages/ipython-0.8.4-py2.4.egg
Extracting ipython-0.8.4-py2.4.egg to /usr/lib/python2.4/site-packages
Adding ipython 0.8.4 to easy-install.pth file
Installing ipython script to /usr/bin
Installing pycolor script to /usr/bin

Installed /usr/lib/python2.4/site-packages/ipython-0.8.4-py2.4.egg
Processing dependencies for ipython
Finished processing dependencies for ipython
```

在具备管理员权限的情况下，Windows系统上的安装过程基本上也差不多。这里是在一台已经安装有最新版IPython for Windows的PC上尝试安装时的结果：

```
C:\>easy_install ipython
Searching for ipython
```

```
Best match: ipython 0.8.2
Processing ipython-0.8.2-py2.5.egg
ipython 0.8.2 is already the active version in easy-install.pth
Deleting c:\python25\Scripts\ipython.py
Installing ipython-script.py script to c:\python25\Scripts
Installing ipython.exe script to c:\python25\Scripts
Installing pycolor-script.py script to c:\python25\Scripts
Installing pycolor.exe script to c:\python25\Scripts

Using c:\python25\lib\site-packages\ipython-0.8.2-py2.5.egg
Processing dependencies for ipython
Finished processing dependencies for ipython
```

B.2 Django

安装完Python后，接下来就轮到Django了。在编写本书时，Django 1.0已经发布，同时1.1版正在开发。本书的内容主要是基于1.0版，所以我们推荐你使用该版本，而且使用最新的稳定版总归是比较保险的。当然如果你很激进的话，下面的信息会帮助你使用Django的开发版本。

打包发行版

Django的打包发行版（Package Releases）可以在项目的网站http://www.djangoproject.com/download/（或者通过你系统上的软件包管理系统）获得。网站上下载的官方版本是以常见的Unix格式.tar.gz打包——Unix和Mac系统可以无须任何设置就能打开它们，而Windows用户则需要额外的软件支持，比如7-Zip（http://7zip.org），命令行下的LibArchive（http://gnuwin32.sf.net/packages/libarchive.htm），或者“Windows上的Unix”环境工具如Cygwin（http://www.cygwin.com）。

开发版本

Django的开发版里包含了稳定版里没有的新特性，它需要用Subversion版本管理客户端才能取得。和Python类似，Subversion通常可以通过软件包管理器（在Unix上）或是MacPorts（在OS X上）获得，或者直接从Subversion的网站（http://subversion.tigris.org）下载也行。

当有了Subversion以后，只需一行命令即可获得Django的最新版本：

```
$ svn co http://code.djangoproject.com/svn/django/trunk django_trunk
```

安装

解开.tar.gz文件（对打包发行版来说）或检出Django的源码（对开发版来说）后，在你当前的位置上会有一个新目录叫做Django-1.0或是 django_trunk。目录里包含了Django的完整代码——不仅有框架（Python模块本身，django目录），还包含了文档以及测试文件等一系列其他脚本和信息。

要使Django工作起来有三种方法：

- 把这个新目录添加到你的PYTHONPATH里去。例如，如果你把trunk检出到/home/username/下的话，你应该把/home/username/django_trunk（而不是django子目录）添加到PYTHONPATH里去。
- 将django子目录移动，复制，或者链接到Python下的site-packages目录里去。
- 在新目录下（以管理员身份）执行python setup.py install命令，它会自动将Django安装到适合的位置。

要知道更多关于Python路径机制的信息，请参阅第一章。我们建议如果你打算使用Subversion来跟踪开发版最新进展的话，第一或者第二种方法都可以（第二种方法里只能用链接的方式）。虽然在trunk里运行setup.py install也可以，不过每次从Subversion更新后你都要再次运行这个命令，所以我们不推荐这种方式。

测试

在Python解释器里简单地输入import django即可检查Django是否正确安装到PYTHONPATH里去了。如果一切正常的话，安装就完成了！如果你得到一个ImportError: No Module named django错误信息的话，重新检查你的步骤，或者试试别的方法。

B.3 Web服务器

安装完Python和Django之后，离使用Django就算是前进了一大步了。下一个重要的步骤是Web服务器，它的作用是把你动态生成的HTML页面返回给浏览器。Web服务器还负责处理图片和CSS等静态的内容，以及各种系统级别的事情（负载平衡，代理等）。

内置服务器：不能用于发布

实际中使用的最简单的Web服务器是Django内置的“runserver”服务器，也叫开发（“dev”）服务器，这是一个基于Python内置的BaseHttpServer的服务器（一个绝佳的利用Python标准库的例子）。在第二章中你已经见过它了，它可以通过manage.py方便的测试一个简单的Web应用。这就是开发服务器的特别之处：快捷地实验任何新特性。用它来调试也是相当的不错，因为它运行在终端里，故而可以查看到Python所有print语句的输出。

但是，要构建一个高效、可靠、安全的Web服务器不是一项简单的任务，明智的Django团队决定不涉及这个领域。开发服务器没有经过大量的测试，所以它绝对不应该被用于任何测试以外的情况，例如部署到Internet上。Django的文档在说到这个问题时半开玩笑地说：“如果你真的考虑要把它用在发布环境里的话，我们就要吊销你的Django执照。”这一点本身是相当严肃的。

最后，虽然开发服务器确实可以处理静态文件（参见官方文档或者withdjango.com），我们强烈建议当你的开发走到这一步时，你最好还是花点时间设置一下下面要提到的服务器环境。而且更好的是，你还可以换到一个更加接近真实的环境进行开发——这有助于及早发现部署上的问题。早发现，早解决。

标准做法：Apache和mod_python

Apache Web服务器和它的mod_python模块一直都是Django站点推荐的部署方式。这也是开发Django的Lawrence团队在他们繁忙的网站上所采用的组合，时至今日，这依然是经历过最完整的测试以及文档最丰富的部署方案。

如果你的情况不允许使用mod_python的话（比如在一个共享的hosting环境里，或是一个非Apache的服务器），请参考下面一节。但是如果你有服务器（或是虚拟服务器实例）的控制权，或是有稳定的mod_python支持的话，这依然是最稳妥的做法。

Django 需要Apache 2.0或2.2以上（有的软件包管理器用apache2来和更古老的apache1.3区分开来）和mod_python 3.0版本以上。和Python的情况类似，具体情况根据平台的不同会有所变化，比如Mac 10.4之前的用户就需要用MacPorts来更新（只有Leopard才支持Apache 2）。Windows用户分别可以在http://httpd.apache.org和http://www.modpython.org找到编译好的二进制版本。

安装完毕后的设置要考虑两个主要的问题，当通过mod_python部署Django时，在哪里连接Django以及在哪里处理静态文件。

在Apache中挂接Django

首先要决定的是Django要处理多少域名的URL——是整个站点（如www.example.com），还是其中的一个或几个部分（如 www.example.com/foo/，这里www.example.com/或www.example.com/bar/会由其他的如PHP或是静态HTML来处理）。我们还可能用到多个Django项目，每一个都有自己的部分。

要挂接一个Django项目需要在Apache中作出如下配置，如果不是虚拟主机的话，通常这个片段是在<VirtualHost>里，或者也可以是你自己的apache2.conf（有些系统上是httpd.conf）里。

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug On
</Location>
```

上面的<Location>会让Django的mysite处理任何由配置文件控制的顶极域名。如果希望Django覆盖URL空间中的一部分，只要做出如下更改即可：

```
<Location "/foo/">
```

这里还有好几种情况：一个是当你的Django项目不在全局系统PYTHONPATH上，但是希望在Apache载入Python模块时带上它。这样的话，你只要为之添加一个额外的PythonPath说明就可以了。如果Django项目mysite（包含有settings.py，root URLconf和app目录）的位置是/home/user/django-stuff/mysite/，你需要把它添加到Python路径里去，如下所示：

```
<Location "/">
    SetHandler python-program
    PythonPath "['/home/user/django-stuff/'] + sys.path"
```

```
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug On
</Location>
```

如果想要在同一个域下面添加多个Python项目，你只要定义多个这样的<Location>就可以了。当然，你得告诉mod_python将它们在内存里分开存放，不然的话会发生未定义的行为。这需要你在每一个块里定义一个唯一的（但不是任意的）PythonInterpreter说明。

```
<Location "/foo/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE foosite.settings
    PythonInterpreter foosite
    PythonDebug On
</Location>

<Location "/bar/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE barsite.settings
    PythonInterpreter barsite
    PythonDebug On
</Location>
```

如你所见，使用mod_python来管理Django代码是非常灵活的。要知道更多关于Python和Apache的信息，请参阅http://modpython.org上的mod_python的文档。

给静态内容“留条缝”

现在Apache已经能正确处理Django了，但是还缺了点什么：图片和JavaScript/CSS（还可能有视频或PDF文件等任何网站可以服务的内容）。一般来说这些文件都是放在应用程序里的某个特定的URL下，比如说你的app是/foo/的话，你的图片和样式表（stylesheet）则可以是/foo/media/。但现在如果把Python代码放在了/foo/，我们就需要给这些静态内容让条路出来。

要做到这一点非常简单：你只需在处理Django项目的<Location>下加上另一个<Location>块，告诉Apache让mod_python不要处理某个特定的位置就可以了。

```
<Location "/foo/media/">
    SetHandler none
</Location>
```

有了这个以后，Django代码可以毫无问题地处理/foo/users/请求，但是对/foo/media/images/userpic.gif的请求则会转去查找Apache的document root（这是由你的configuration或是virtual host块定义的）。另外，你完全可以在mod_python的URL空间里开上多个这样的“洞”，就是说你可以为图片、CSS和JavaScript准备不同的目录，如下：

```
<LocationMatch "/foo/(images|css|js)/">
    SetHandler none
</LocationMatch>
```

<LocationMatch>和<Location>不同之处在于它用到了正则表达式，这里我们有一个简单的正则表达式来表明不希望mod_python处理这三个目录。这个和定义三个<Location>块的作用是一样的，不过这种方式要简洁得多。DRY原则不但适用于Python代码，也适用于系统配置！

另一种灵活的选择：WSGI

WSGI（Web Server Gatewat Interface）和mod_wsgi是Python Web hosting技术中正在冉冉升起的一颗新星。Django对WSGI的支持相当得完整，越来越多的Django程序员（还有Python的Web程序员）觉得它比mod_python更好用。WSGI是一种非常灵活的协议，旨在把Python桥接到任何兼容的Web服务器上，不仅仅是Apache，还有lighttpd（http://lighttpd.net），Nginx（http://nginx.net），CherryPy（http: //cherrypy.org），甚至微软的IIS。

虽然WSGI技术还相当新，但是它已经能够在前面提到的所有Web服务器上工作，并且经历了大量Python Web框架的测试，包括Django，和一些流行的独立Python Web应用如Moin Moin wiki引擎和Trac软件管理系统。

mod_wsgi 的主要卖点（除了Web服务器无关这个特点）在于相比mod_python它占用的内存更小性能却更好，所有的WSGI应用有统一的接口标准（包括 Django之外的应用），以及支持守护进程模式，可以轻松地让一个WSGI进程只属于系统上的某个特定用户。

mod_wsgi最大的缺点是（到编写本书的时候）它还没有被软件包管理系统广泛接受，所以很有可能你得自己编译安装它。不过在不久的将来这一现象一定会得到改善的，Windows用户目前则可以使用mod_wsgi官网上链接的一些非官方的Windows二进制包。

如果找不到适合你操作系统的预编译版本的话，你可以从mod_wsgi的网站http://modwsgi.org/ 上获取一份源代码，然后按照下列的步骤自己安装。安装运行的步骤并不复杂，实际上比mod_python还要简单一点。首先，配置Apache及其支持模块。拿Apache 2来说，只要在http.conf里添加一行代码即可：

```
LoadModule wsgi_module /usr/lib/apache2/modules/mod_wsgi.so
```

然后在configuratio块里加上：

```
Alias /media/ "/var/django/projects/myproject/media"
<Directory /var/django/projects/myproject/>
    Order deny,allow
    Allow from all
</Directory>
WSGIScriptAlias / /var/django/projects/myproject/mod.wsgi
```

最后创建一个mod.wsgi的脚本，引用上面的最后一行：

```
import os, sys
sys.path.append('/var/django/projects')
os.environ['DJANGO_SETTINGS_MODULE'] = 'myproject.settings'

import django.core.handlers.wsgi
```

```
application = django.core.handlers.wsgi.WSGIHandler()
```

你需要把上面代码里所有的/var/django/projects替换为你实际的Django项目的路径。比如sys.path.append里添加的是包含你项目的路径，即/var/django/projects，而不是/var/django/projects/myproject。

另一种选择：Flup和FastCGI

我们要介绍的最后一种可行的Web服务器部署方法是Python模块flup，它不仅是另一种使用WSGI的桥接（flup初始的目标），而且还支持另外一种类似的协议如FastCGI（有时简称为FCGI）。和WSGI一样，FastCGI的目标是在用户的应用代码和Web服务器之间架起一座沟通的桥梁，同时它也具有以下优点：由于运行在单独的进程里，所以具有更好的潜在性能；另外以和Web服务器分开的用户身份运行也增加了安全性。

就目前的情况来说，FastCGI在共享hosting平台上比WSGI支持的更好，部分原因是由于它支持多种语言，另一部分则是因为它存在的时间更长。因此，如果没办法使用Apache而你的环境又不支持WSGI的话，FastCGI或许是个很不错的选择。

Django的官方文档上（你可以在withdjango.com上找到链接）有一个非常棒的关于如何设置FastCGI的教程，所以在这里我们就不再重复了。如同文档里所提到的，flup在WSGI和FastCGI之外还支持了SCGI和AJP两种额外的协议，如果你的部署要求不太常见的话，或许这些协议会对你相当的重要。

B.4 SQL数据库

Python代码（Python和Django）已经准备就绪，Web服务器（Apache、Lighttpd等）也准备好处理动态请求了，现在还差最后一步。你需要数据持久化，而这正是SQL数据库的强项。Django可选的数据库很多，通常每种都需要其自己的数据库软件，以及相应的Python接口库配合使用。

除了下面的介绍之外，Django的官方文档里还专门给出了一些关于各种平台上常见的问题（可以在withdjango.com上找到链接）。如果你遇到数据库选择上的问题的话，可以先看看这份资料。

SQLite

SQLite 是一个名副其实的“轻型”SQL数据库实现。和PostgreSQL、MySQL，以及各种商业数据库如Oracle或MS SQL等不同的是，SQLite不是以一个独立的服务器的形式运行的，它只是提供了一个访问磁盘上数据库文件的接口库。和其他复杂服务的“轻型”实现一样，SQLite也有优点（易用，低耗）和缺点（功能少，数据量变大时性能较差）。

所以，如果不想在你的小站点上引入完整的数据库服务器的话，SQLite正适合你的需求（就像你在第二章中已经看到的一样）。不过一旦你过了学习阶段进入正式部署阶段后，最好还是升级成更加适合的数据库。

和SQLite交互需要一个对应的Python库。如果你使用的是Python 2.5，那么它已经内置支持了sqlite3模块，如果是Python 2.4或更旧的版本，那么可以访问 http://www.initd.org（或者你的软件包管理系统）来获取pysqlite模块。

和其他数据库不同，SQLite无须显式地创建数据库，也没有什么用户管理。只需告诉它数据库在文件系统上（Web服务器可以读写）的位置并且将其记录在你settings.py的DATABASE_NAME选项里就可以了。设置完成后，常用的Django工具如manage.py syncdb就可以往那个路径里写入一个SQLite数据库文件了。

SQLite的SQL shell命令是sqlite3（有时候就是sqlite，特别是当你的系统只有SQLite 2.x的时候）并且接受一个数据库文件作为参数，例如sqlite3 /opt/databases/myproject.db。

PostgreSQL

PostgreSQL（或者简称为“Postgres”）是一个功能完善的数据库服务器，它提供大量丰富的特性，并且作为业界领先的开源数据库之一拥有出色的声誉。Django核心团队推荐使用的就是这款数据库，并对其质量颇多赞誉。Postgres目前的第8版（不过Django也能支持第7版）支持了所有的主流平台，虽然它和 MySQL比起来在共享服务器上不是那么流行。

Postgres的官网是http://www.postgresql.org，如果你在Windows系统上或是在软件包管理系统里找不到的话，可以去http://www.postgresql.org/ftp/binary/下载，找到最新的版本，然后选择你的平台即可。

Python 要使用Postgres的话需要psycopg库的支持，最好是用第2版，有时也叫psycopg2。psycopg可以从 http://initd.org/pub/software/psycopg/或是从软件包管理系统里获得。这里要注意的一个小地方是Django有两个不同的数据库后台，分别对应不同版本的psycopg。小心确认你用的是正确的版本！

在Postgres里创建数据库和用户十分方便。默认安装包括了独立的命令行工具如createuser和createdb，从字面上就能知道它们的作用了。根据安装系统和安装方式的不同，数据库可能已经为你创建了一个超级用户的账号——有时它是一个postgre系统级的用户，有 时则是你自己的用户名。请参阅相关文档来确认这一点——官网上的或是你自己系统里的都行（如果你是通过软件包管理系统安装的话）。

得到Postgres超级用户的用户名和密码后，你可以参照下面的例子来为你的Django项目创建一个新的数据库和用户：

```
$ createuser -P django_user
Enter password for new role:
Enter it again:
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) y
Shall the new role be allowed to create more new roles? (y/n) n
Password:
$ createdb -U django_user django_db
Password:
```

上面的例子创建了一个带密码的Postgres数据库用户django_user并且赋予它创建新数据库的权限，随后利用该用户创建了一个名为 django_db的新数据库。在完成上述步骤并将用户名和数据库名（以及用户密码）添加到settings.py里以后，你就可以使用 manage.py来创建和更新数据库了。

注意例子里最后的那个"Password: "是Postgres提示我们输入数据库的密码——我们的系统用户同时也是数据库的超级用户。否则我们需要用-U选择运行createuser来指定超级用户的用户名，就像调用createdb时做的那样。

最后，PostgreSQL的SQL命令行程序是psql，它有很多和createdb与createuser一样的选项，比如用-U指定用户名。

MySQL

MySQL是另一种主流的开源数据库服务器，最新的版本是5（也支持版本4）。虽然MySQL缺乏一些Postgres的高级功能，但是相比起来它更流行一些，这主要是因为它和常用的Web语言PHP的紧密结合。

和其他数据库不同，MySQL在内部有几种不同的数据库类型来支持不同的特性：一种是MyISAM，缺乏事务支持和外键但是可以进行全文搜索；另一种是 InnoDB，它比MyISAM更新，特性更丰富，但是目前缺乏全文搜索能力。虽然还有其他的一些类型，但这两个是最常见的。

如果你在 Windows上或你的软件包管理系统没有最新版MySQL的话，它的官网 http://www.mysql.com 提供了大多数平台的二进制版本。Django推荐的MySQL Python库是MySQLdb，可以在其官网 http://www.sourceforge.net/projects/mysql-python上下载到，你需要的是1.2.1p2以上的版本。请注意这里的版本号，有些旧的Linux发行版自带了如1.2.1c2这样的版本，Django无法兼容这样的旧版本。

MySQL里创建数据库的工作主要是通过全能的admin工具mysqladmin来完成。和Postgres一样，你需要先弄清楚安装版的数据库超级用户名和密码才能为Django项目创建新用户。这个超级用户通常是root，并且没有初始密码（最好尽快修改掉），所以创建数据库非常地方便：

```
$ mysqladmin -u root create django_db
```

和Postgres不同的是，MySQL的用户完全由数据库自身管理，所以我们要用MySQL的SQL shell来创建数据库用户django_user。这个shell的名字是mysql。

```
$ mysql -u root
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6
Server version: 5.0.51a-6 (Debian)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

```
mysql> GRANT ALL PRIVILEGES ON django_db.* TO 'django_user'@'localhost' IDENTIFIED
BY 'django_pass';
Query OK, 0 rows affected (0.00 sec)

mysql>
```

经过以上几个步骤，你现在就可以修改settings.py的设置并开始使用manage.py执行数据库相关命令了。

Oracle

在编写本书时Django目前支持的最后一个数据库是一个商业数据库，Oracle。如果你还是一个数据库驱动开发的新手的话，基本上你不用考虑这个选择，因为Oracle在共享主机上不太常见，也无法通过Linux软件包管理系统得到。不过，Oracle是有免费版的，Django可以支持9i以上的版本。

连接Oracle的Python库是cx-oracle，可以从 http://cx-oracle.sourceforge.net/ 获得。

其他数据库

Django 的数据库支持还在不断的进化中。获得最新信息的最佳来源是官方网站和Django的用户列表。Django目前没有直接支持的两个数据库是微软的SQL Server以及IBM的DB2。Google Code上有一个独立维护的项目为Django提供了对MS SQL的支持：http://code.google.com/p/django-mssql/。另外Google Code上还有一个支持DB2的项目Python-DB2 http://code.google.com/p/ibm-db/，在编写本书的时候，这个项目对Django还不是即插即用的。

B.5 总结

到这里，我们应该已经完整地配置运行了Django及其每个组成部分——Python，Django，Web服务器和数据库。如果你遇到了任何问题，请回忆一下我们在第二章里分享的建议——首先备份你的工作，重复所有的步骤，确保你没有遗漏文档中的任何一步。

最后，Django自己的安装文档（不仅有Python库，还包含了所有的组件）也是一份非常出色的资料，你可以在withdjango.com上找到链接。

附录C 实用Django开发工具

再小规模的Web开发也是软件开发。只要你的站点接受某种形式的用户输入并且处理这些输入，它就算是一个Web应用。它和（或者说应该和）其他任何类型软件的开发都是相似的。

如果你是从软件开发转入Web开发来的话，那对你来说使用现有的软件开发工具和技术都是秃子头上的虱子——明摆着的。这些工具有版本控制，bug追踪，以及强大的开发环境等。如果是这样的话，你只需要快速浏览一下本节的内容就行了，确认没漏掉任何能帮助你提高工作效率的东西。

如果你是来自设计或别的领域，这里介绍的内容对你来说可能是全新的体验。好消息是，我们保证只要你花功夫学习掌握这些工具，就可以成倍地提高生产力以及获得极大的灵活性，绝对能让你省不少心。

C.1 版本控制

如果你在开发任何形式的软件，而又没有使用版本控制的话，就实在太落伍了。版本控制系统为你的项目保留了一个完整的修订历史，允许你将代码回溯到历史上的任何一点（比如，回到一个星期前的某个看似无害的改动发生前的一个小时）。

早期的版本控制系统如SCCS（Source Code Control System）和RCS（Revision Control System）相当简陋，要么是维护一份原始版本的文件加上些许修改（不同版本之间文件的微小改动），要么就是反过来——维护一份最新的版本，然后减去新的改动。这类系统的一个主要问题是被控制的文件和修改都在同一个服务器上。

随着软件开发的不断进步，特别是在开源社区里，大家都意识到这种系统很不适合分布式的团队开发，所以就出现了更现代的版本控制系统，例如RCS的改进型和分布型分支，CVS（Concurrent Versions System），以及后来的Subversion项目，Subversion希望能成为一个“更好的CVS”并最终取代之。

主干和分支

版本控制系统往往看起来像一棵树，其中开发的主线被称为主干（trunk），而在这之上所作的复制（于是变成了独立的实体）会按照它们自己的方向发展下去。这些拷贝就是分支（branch），如何在主干和分支上分配任务则取决于项目本身。

一种做法是把所有新特性的开发（即会打破向后兼容性或是引入不稳定性的开发）放在主干上，用分支来表示软件发布（release），从主干分出来后，分支只接受bug修正。另一种方式则完全相反，保持主干稳定，把所有的新特性放到分支里去，这样做的好处是可以同时开发好

几个庞大的新特性。

Django自己用的是一种介于上面两者之间的方法：它同时维护了两条分支，一条是发布，另一条是特性，而主干则保持折衷——既不是完全稳定也不是完全不稳定。会产生严重影响框架稳定性的改动会在分支里完成，而小修小补则直接在主干上进行。

合并

在主干上靠剥离分支对隔离代码库拷贝很有用，但是要是没法将分支上的修改合并回主干的话，分支的作用也就不过如此了！这里要稍微打破一点树的形状：源码控制里的另一个主要概念就是能将一个分支里的修改合并到另一个分支里去。

例如，Django的表单框架最近要进行一次大翻新，由于涉及相当多的改动，所以准备放到一个单独的分支里。稍后当任务完成时，我们得到了一个版本更优的框架。在这期间，主干在各个部分仍然在不断地更新，就是说不单这个“新表单”分支在原有的基础上作出了一大堆修改，而且主干在这段时间里也有一份长长的修改清单。

关于如何调和这类改动集合的理论有很多，但基本一点是版本控制工具要提供命令来将这类修改合并起来，并加到某一个分支上去。拿 Django来说，维护人员通过运行一些命令来把“新表单”分支里的改动更新到主干里去，同时手动处理一些版本控制程序无法调停的矛盾就可以完成合并了。

明确版本控制的职责之后，现在我们来看看现在主流的两种源码控制技术，并简要的介绍一到两种特定的版本控制系统。

集中式的版本控制

开源集中式的版本控制（open source centralized version control）里的两个大明星分别是CVS和Subverion——后者正在慢慢取代前者。更旧一点的系统包括RCS和SCCS。商业系统则包括 Perforce，IBM Rational ClearCase和微软的Visual Studio Team System。

Subverion

Subverion（http://subversion.tigris.org/）是目前最接近工业标准的版本控制系统。它是一个开源系统，能运行在很多平台上，拥有良好的性能和稳定性，并且已经被业界广泛接受。

Subverion 按照集中式的方式进行操作，它有一个主仓库（master repository）供所有的用户连接。你检出（checkout）的代码可以在没有网络连接的地方工作（即修改代码），但是要把改动返回仓库或是从仓库里获得更新都需要你在线才行。Subverion会记住自上一次从中心服务器获取更新后所作的一切改动，但如果没法连接数据库的话，它是没法同时记住多个改动集的。

Subverion是目前Django主代码仓库（位于http://code.djangoproject.com），以及很多Google Code上Django相关项目所采用的版本控制系统。Subverion还能无缝地和Trac Wiki以及issue-tracker集成（Trac会稍后介绍）。

分散型的版本控制

分散型版本控制（decentralized version control）是未来的趋势所在。一个分散式系统可以完成集中式系统所能做到的一切，而且还具有更强大的的特性，即项目的每个“checkout”自身也是一个完整的仓库。在这种分布式的版本控制之下，你不再需要连接到中心服务器来记录每一组改动集。本地仓库就能帮你记住它们。只有在需要的时候，它们才会通过网络传递出去/进来。

开源的分散式版本控制系统有Git，Mercurial，Darcs，Bazaar，SVK和Monotone。商业系统则包括BitKeeper和TeamWare。

Mercurial

Mercurial（http://www.selenic.com /mercurial/）是今天最流行的分布式版本控制系统之一。Mercurial主要是用Python编写，并加上一些C代码（在特别需要性能的部分）。Mercurial的工程师把性能看得非常重要，所以很多大型的项目都采用了Mercurial，其中包括Mozilla浏览器和Sun的OpenSolaris系统。

Git

Git（http://git.or.cz）算是Mercurial的竞争对手，它们和 Bazaar一起瓜分了开源DVC（分布式版本控制系统）的份额。Linus Torvalds和其他Linux内核团队的成员一起用C编写了这个软件。Git是一个相当符合Unix精神的工具，原本是创造出来帮助处理Linux内核项目这种极端复杂的源码控制需求的。Ruby on Rails等其他很多Rails影响下的项目，以及WINE项目和X.org（Linux的图形系统），Fedora Linux等都不约而同地使用了Git作为其版本控制系统。

为你的项目添加版本控制

以下为不知道怎么在Django项目里使用版本控制的读者提供了一些基本指导。在这里我们用的是Mercurial（hg命令）。

首先我们建立一个Django项目框架。

```
$ django-admin.py startproject stuff_dev_site
$ cd stuff_dev_site
$ ./manage.py startapp stuff_app
$ ls stuff_app
__init__.py   manage.py    settings.pyc  urls.py
__init__.pyc  settings.py  stuff_app
```

现在把这个工作目录添加到仓库里去。

```
$ hg init
```

接着添加一个基本的.hgignore文件（你可以像下面那样用echo命令，也可以用文本编辑器），告诉Mercurial略过那些后缀名是.pyc的字节码文件，这些文件都是由Python的字节码编译器自动生成的，我们没必要跟踪它们的历史。

```
$ echo "\.pyc$" > .hgignore
```

在默认情况下，.hgignore文件里的模式都是由正则表达式组成。接着我们来添加文件：

```
$ hg add
adding .hgignore
adding __init__.py
adding manage.py
adding settings.py
adding stuff_app/__init__.py
adding stuff_app/models.py
adding stuff_app/views.py
adding urls.py
```

然后，用hg commit命令提交修改，并确认Mercurial在hg log里有把它们记录下来。

```
$ hg commit -m "Initial version of my project"
No username found, using 'pbx@example.org' instead
$ hg log
changeset:   0:e991df3d3205
tag:         tip
user:        pbx@example.org
date:        Sun Oct 07 13:49:14 2008 -0400
summary:     Initial version of my project
```

每次向Mercurial仓库提交代码时，Mercurial都会用两个不同的数字来标记一个改动集(changeset)：一个只在本仓库里有效的自增整数和一个十六进制的哈希数（标记了它在主仓库里的ID号）。

现在我们来做一点小修改，看看发生了什么，然后提交它。

```
$ echo "I_LIKE_CANDY = True" >> settings.py
$ hg status
M settings.py
$ hg commit -m "Apparently someone likes candy."
No username found, using 'pbx@example.org' instead
$ hg log
changeset:   1:65e7cda9f64b
tag:         tip
user:        pbx@example.org
date:        Sun Oct 07 13:57:53 2008 -0400
summary:     Apparently someone likes candy.

changeset:   0:e991df3d3205
user:        pbx@example.org
date:        Sun Oct 07 13:49:14 2008 -0400
summary:     Initial version of my project
```

当记录下一个改动集时，它会作为项目在当时的“快照”被保存下来。如果你发现在改动集0所作的改动之后完全把事情搞砸了的话，只需要用命令hg revert --rev 0 -all将工作目录回滚到那一点上去就可以了。

如此这般有条不紊地继续进行项目。修改代码，在开发服务器上测试，用简单明了的提交

信息记录修改。接着就要准备部署你的项目了。

你可以把所有的文件打包起来，复制到要部署的服务器上，然后解开文件就行了。如果你从此不再对这个应用做任何修改的话，这种方式也算不上很糟糕。但事实你开发的是软件，不是大理石雕像，将来修正和增强这个项目的可能性非常高，这种重复打包－复制－解包的过程太麻烦了。

更好的办法是直接从仓库里复制一份拷贝出来，同时让Mercurial记住拷贝的出处，因为这样只要递增地获取任何在这份拷贝上的更新就可以了。这正是hg clone命令所完成的工作。

为了简单起见，我们假设部署到网站上的拷贝和开发中的拷贝都在同一台服务器上。这样一来事情就变得相当简单，只需从当前工作目录转到要创建拷贝的地方，然后输入hg clone和原始（开发）目录的路径就行了。比如说，我们现在在原始目录下，希望在同一级目录下创建一份部署版本的拷贝，那么相应的命令就是：

```
$ cd ..
$ hg clone stuff_dev_site stuff_live_site
8 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

这就会对原始的仓库生成一份和工作目录一模一样的拷贝（clone）。

```
$ ls stuff_live_site/
__init__.py  manage.py  settings.py  stuff_app  urls.py
```

等一下，那些.pyc文件都到哪里去了？因为前面我们告诉过Mercurial让其忽略掉它们，所以这些文件并没有被加入到仓库里去，所以当然也就不会显示在这里了。

现在我们在原始（开发）目录上作一点修改。

```
$ cd stuff_dev_site
$ echo "I_LIKE_DOGS = True" >> settings.py
$ hg commit -m "Also, dogs are liked."
```

当充分测试这些修改并准备好部署时，我们只要转到“live”分支下，执行以下命令把那些改动加进来即可：

```
$ cd ../stuff_live_site
$ hg pull -u
pulling from /stuff_dev_site
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

在live站点上也可以用相同的hg revert命令来放弃之前的改动——比如我们发现刚刚加入的改动虽然通过了dev站点的测试，却在live站点上产生了预料之外的问题。项目的每一份克隆都有相同的内容和历史。

这里介绍的只是一些皮毛，不过希望至少能让你有一点概念。当然了，版本控制里，特别是在Mercurial里，要学的东西比这里展示的还要多很多。更详细的信息，包括免费的Subversion、Mercurial等系统的手册，请参见withdjango.com。

C.2 项目管理软件

用版本控制系统管理代码是很有用，不过有些程序员却到此为止了。其实很多其他程序员还用到了一种经常和版本控制系统“搭档”的Web应用程序，它不但提供了源码仓库及其历史的Web界面，同时还能跟踪问题，TODO条目，以及文档等。

这样的软件包市面上有很多，其中大多是以服务的形式出现，例如Google Code，SourceForge，Launchpad和Basecamp。其他的则是一些可以自行安装的独立应用程序。开源社区里用的最多的一个就是Trac，它也是用Python编写完成的。

Trac

Trac是一个开源的wiki，bug跟踪，源码和项目管理系统，它由Edgewall Software公司负责维护。我们承认这里有一点偏激：不过Trac是我们在项目管理软件领域里最喜欢的一款产品。它无须太多设置即可完美运行，特别是能和Subversion等版本控制系统很好地集成在一起。它还有一些常用的wiki标记可以用来轻松地链接源码版本，bug记录和文档笔记等内容。

要是这些还不够的话，你要知道Django代码仓库本身 http://code.djangoproject.com/ 也是运行在Trac之上的。（只不过由于那些漂亮的自定义风格，所以你没有认出来吧。）

> **注意**
>
> Trac是编写本书过程中的主要工具——更多信息请参考本书扉页。

另外，虽然Trac默认支持的是Subversion，但是也可以通过插件让Trac支持其他系统（比如前面提到的那些）。你可以在 http://trac.edgewall.org/下载到Trac以及阅读它的完整文档。如果你需要扩展Trac的话，可以先看看 http://trac-hacks.org/上有没有适合的插件和“hack”技巧。

C.3 文本编辑器

编写Django项目不需要任何特殊的软件。任何程序员的编辑器都可以胜任这一工作。以下是一些常见编辑器里的小技巧。

Emacs

作为一个高效使用Emacs的Django程序员，你手中最重要的武器就是python-mode，它具有语法高亮，自动对齐，以及其他很多提高生产力和方便编写Python代码的功能。Emacs第22版以上都内置了python-mode。对于旧一点的版本，或是Emacs的其他变型如XEmacs等，可以访问Python官方网站上的Emacs页面（http://www.python.org/emacs/）。

另外你还可以在Django的wiki上找到为Django模板所准备的第三方mode(http://code.djangoproject.com/wiki/Emacs)。

Vim

Vim 编辑器来自过去UNIX下的工具vi。它大大增强和改进了原有的工具。你可以在vim的主页http://www.vim.org/上下载到它。和Emacs一样，vim也有Python语法mode。Django wiki上可以找到一个专门介绍如何在vim（以及它的变型）里使用Python的页面。

TextMate

TextMate是 OS X下非常流行的一个商业编辑器，它对Django的支持非常出色。TextMate把它对某种语言和语法的支持组织成一个个“插件”(bundle)，其中的Python插件能大大加速Python代码的编写，并通过代码着色令它更容易阅读。此外，在TextMate公共的插件仓库里还有两个专为 Django准备的插件：一个是为Python准备的，另一个则是给Django模板准备的。更多的信息可以在Django wiki上找到（http://code.djangoproject.com/wiki/TextMate)。

Eclipse

Eclipse IDE提供了一个强大的Python开发模块PyDev。你可以从它在SourceForge上的页面里获取更多的信息和代码（http://pydev.sourceforge.net/)。

附录D 发现、评估、使用Django应用程序

编写Django应用是一件很有意思的事情，但是有时候没必要重新编写。随着框架越来越流行，一系列开源的Django应用如雨后春笋一般出现。这得益于Django自己就是一个开源项目，作为榜样，它树立了一个典范，即有问题的时候，不妨将应用源码开放。

更棒的是，大多数这些应用都沿着Django的脚步采用了限制很少的BSD/MIT风格的版权协议。这些应用因此更容易被不方便使用大多数开源协议（GNU Public License或GPL）的组织内部接受。GPL要求如果一个基于GPL协议的产品用在一个会被重新发布的应用程序（相比只是通过网络提供服务）里的话，这个应用程序的代码也必须公开。BSD/MIT风格的协议虽然也支持这一点，但却不是必须的。这里没有任何要对两种协议品头论足的意思，只不过事实上BSD/MIT风格的协议对商业应用更友好，在你作出选择时必须要将协议的问题作为考量之一。

无论是简单到用户注册这种功能，还是复杂如完整的blog引擎，或是运行你网络商店的整套电子商务解决方案，基本上一定已经有一个开源的Django应用可供你使用了。但是要怎么找到它呢？怎么判断它的质量？以及是否可以尽量简单地使用（并持续更新）它？

D.1 到哪里去找应用程序

有些程序员拥有自己的网站和项目管理系统，因此你经常会发现他们把Django应用列在blog或是个人Trac系统里。不过，更多的程序员选择将项目放到集中的列表网站上以期获得更高的访问量，这里我们列出几个流行的网站。

- Django项目wiki上的DjangoResources页面：很多应用程序的作者选择把作品放到这里来展示给全世界，它的连接为 http://code.djangoproject.com/wiki/DjangoResources。
- Google Code：免费的站点和干净的界面，让Google Code迅速成为放置Django项目趋之若鹜的地方。在项目查找里输入django就能得到一份长长的列表。
- Djangopluggables.com：在编写本书的时候，这还是一个相对较新的站点，它收集了各种关于Django应用程序的信息。
- DjangoSites.org：这个简单的Django站点有一个“sites with source”类别可以选择只显示那些开放了源码的项目。
- GitHub.com：这是一个使用Git版本控制系统的源码仓库和社交网络站点，它也包含了一些Django应用程序，并随着Git变得越来越流行。

由于Django社区越来越壮大，寻找Django应用的方法也越来越多。经常到withdjango.com看看更新的链接和推荐吧。

D.2 如何评估应用程序

下面这些问题是当你找到一个预期的应用和项目时要确认的问题。

- 它是不是够活跃？最近一次的特性更新或bug修正是多久之前？虽然不是每个项目都要一周更新好几次，但是如果一个项目最后一次可见的活动是一年前的话基本上可以认定它已经被抛弃了。任何pre-1.0版之前的Django项目都需要及时更新，否则很快它就没用了。
- 文档是否完整？或者到底有没有文档？即使有，是不是组织良好、容易阅读？有没有很多类似“注意：这一节需要更新！”的警告标志？是不是有人定期维护？（例如，文档是不是通过版本控制管理，并且作为项目的一部分提供下载？）
- 作者是谁？Google一下作者的名字看看他们的经验水平如何，以及他们的工作和与社区的关系。如果你能在项目的邮件列表里得到许多有用的回复的话，这就是一个好现象。
- 代码质量如何？如果你是一名有经验的Python程序员，感受项目最好的办法之一就是下载然后读一下它的源码。源文件组织的怎么样？函数和方法里有没有docstring解释它们的意图和用法？有没有测试，而且是不是都通过了？
- 有没有社区支持？很多Django项目一开始只是为了解决某个特定个人或项目的某个特定需求，然后慢慢培养出自己的用户群和开发社区。应用程序越是复杂，其背后越是可能有一个由资深用户组成的社区提供帮助。当然，不是说每个应用程序背后都一定要有一个活跃的社区才算是可用的，但是社会基础也可以在一定程度上反应出应用程序的复杂程度。

D.3 如何使用应用程序

第三方的（包括你的）Django应用程序，其实就是Python模块。要在一个项目里使用它们，只需要在你项目的settings.py文件中的INSTALLED_APPS设置里加上一个包含应用程序的路径的字符串就行了。

虽然你可以把它们放到任何地方，不过一般来说有三个可选项：

- 内嵌：如果只是拿它作为一个单独的项目，你可以选择把它加到你项目的目录下和其他应用程序放在一起。可以说这是最简单的方法。缺点是当要在同一个服务器的其他项目里使用它时，你得粘贴复制。
- 创建一个“共享应用”目录：另一个选择是为这些共享应用单独建立一个目录（比如shared_apps）然后把它添加到Ptyhon路径里。这样所有第三方的代码就被保留在同一个地方，在跨项目导入时十分方便。要把应用程序添加到一个给定项目时，只要把它的名字（无需额外前缀）添加到项目的 INSTALLED_APPS设置里即可。
- 安装到你的site-packages目录：你还可以把Django应用添加到系统范围中的Python库里，这通常是一个名为site-packages的目录。（在Python提示符下输入import sys; sys.path就可以显示你系统的路径，以及Python路径里的目录。）

如果你要跟踪一个不断演化的项目，你可以选择直接从项目的版本控制系统里获取代码而不是下载打包文件。这样的话，根据你选用的不同方式，只需把最新的代码检出（check out）到你的项目、共享应用目录，或是系统site-packages下就行了。记住在更新这类外部应用的时候要格外小心，不要破坏依赖它们的项目。

D.4 分享你自己的应用程序

作为一个不断进步的Django应用程序员，总有一天你会发现你的一些成果可以用来帮助别人。我们强烈推荐你在这种情况下考虑用开源协议开放这些应用的源码，让其他Django用户能使用并改进它。Google Code，SourceForge，GitHub以及其他的代码托管服务都可以让你轻松地和他人分享代码（还不用自己创建一个新的网站来管理）。到时候记得告诉我们你的成果哦！

附录E 在Google App Engine上使用Django

这节附录将会介绍如何把你的Django应用移植到Google App Engine上去，这是一个部分基于Django且内含很多Django特性的可扩展Web应用平台。虽然我们会尽量覆盖所有的基础，但是要详细讨论这一技术的方方面面是不现实的。你可以在本节的最后找到更多关于这些信息的链接。

这里根据包含“Django元素”的多少，可以分出几种为App Engine开发应用的方法：

- 全新的纯App Engine应用
- 将一个现有的Django应用移植到App Engine上去
- 专门为App Engine编写新的Django应用

第一个方式只（最少限度的）使用每个App Engine应用所提供Django特性和组件——我们在下一节里讨论这个。另外两个方式引入了更多的Django特性，这才是本附录主要关心的内容。如果你希望了解更多关于“纯”App Engine应用的内容，请参考最后给出的相关链接，这里我们不多做介绍。

我们主要关心的是Django——为Google App Engine自身开发应用可以写成另一本书了！

E.1 为什么App Engine如此重要

Google App Engine对那些正在使用和打算使用Django的人来说绝对是一个利器。它在广大的软件开发社区里对Python和Django进行了一次很好的宣传，引起了很多以前从未考虑过这些技术的人对它们的兴趣。

可能App Engine作出的最重要的承诺就是它让部署和服务器维护不会再成为一个问题。我们和Web服务器设置打交道（部署Apache或nginx，是用mod_python还是FCGI等），是为了要让网站能运行Python，我们别无选择。

这和PHP的情况不同（虽然PHP自身也有很多问题，但至少语言本身不是），通常它不用担心这类问题。App Engine的出现让Python的Web工程师也能享受到这种好处，即不必再关心编程平台的问题了。而且，它还让我们能利用Google现存的（巨型的）基础设施。

E.2 纯Google App Engine应用程序

纯Google App Engine应用程序一般创建保存在单个目录下，用App Engine提供的工具包进行开发：这包括SDK的开发服务器（dev_appserver.py）以及应用程序上载程序（appcfg.py）。当你的代码准备好发布时，就可以用它把应用程序部署到App Engine的“云”上去。

App Engine应用程序的配置被定义为一个YAML文件（app.yaml），它看起来如下：

```
application: helloworld
version: 1
runtime: python
api_version: 1

handlers:
- url: /.*
  script: helloworld.py
```

你可以看到这里的handler一节和Django的URLconf作用相似，它将一个类URL的字符和对应要执行的脚本挂钩。App Engine开发里的另一个和Django类似的特性就是它的模板系统，它完全照搬了Django的实现。

E.3 App Engine框架的局限性

从 Django程序员的角度来说，Google的实现里缺少的最重要一环就是Django的ORM。Google依赖的是它自己的BigTable存储系统而非关系数据库——你可以在 http://labs.google.com/papers/bigtable.html 和 http://en.wikipedia.org/wiki/BigTable 找到更详细的信息。这里没有SQL语句，没有关系，也没有JOIN。

虽然你可以用Django的其他组件并且把数据模型替换为基于Google BigTable的ORM来编写全新的应用程序，但是这一限制已经基本上把所有现存的Django应用排除出去了。虽然你也可以重写应用程序里相关的部分来绕过它，但是还有很多其他的Django应用和组件是你不想碰的，比如admin，认证系统，通用视图等。

在下一节里，我们会讨论如何为App Engine改写你的应用程序。

除去Django的这些部分，剩下的就是其核心功能了：URLconf，视图和模板。虽然用这些组件也足够构建起任何类型的网站了，但是如果要在App Engine上使用现存的Django应用或是编写能同时部署在普通服务器和App Engine上的应用，这些组件就完全不够了。

> **注意**
>
> 在本书编写的时候，App Engine提供的Django组件是有点落伍不过还算稳定的Django 0.96.1版。

E.4 Google App Engine Helper for Django

Google App Engine Helper for Django是让你感觉到使用App Engine就是“真正”在进行Django开发的关键所在。这是一个由Google赞助的开源项目（Python之父Guido van Rossum也是作者之一），旨在为那些有经验的Django工程师在App Engine上提供一个更熟悉的环境。你甚至可以用它把App Engine附带的Django替换成最新的版本。

获取SDK和帮助程序

首先，我们要下载必要的软件。你可以在SDK项目的主页 http://code.google.com/p/

googleappengine/ 下载到适合你平台的Google App Engine SDK。Windows用户可以下载.msi文件，而OS X用户则可以下载.dmg文件，其他平台的用户则可以下载普通的源码ZIP文件。类似地，帮助程序可以在 http://code.google.com/p/google-app-engine-django/ 获取。

根据向导分别安装它们，完成后你就可以开始使用App Engine了。例如，我们建议先跟着教程构建一个简单的应用程序。Google提供了完备的文档来帮助你设置和运行App Engine，所以这里我们就不重复了。有关App Engine的教程请见 http://code.google.com/appengine/docs/gettingstarted/。

了解一下如何分别在两个平台创建简单的应用是很有好处的，这样当我们尝试将Google App Engine引入现有的Django应用或是以Django的方式构建全新的App Engine的时候，你会觉得容易一点。

再论帮助程序

在本书编写的时候，帮助程序（helper）还处于初始的阶段。它的主要目标是消除Django程序员对App Engine的陌生感，要完成这一点还有很多工作要做。如果想看它的源码的话，你得准备好学习一点App Engine和Django内部的原理。

这个帮助程序不是要把App Engine变成Django，它的作用是让过渡更加平滑。虽然下面的例子假设你用到了帮助程序，不过直接使用App Engine的功能也是完全可以的。就像我们谈论Django时常说的，“它其实就是Python而已”。不要因为App Engine复杂的表象就停止你探索的脚步。有人说，App Engine“想要不扩展都难”。如果你愿意为部署的便捷性及其潜在的无限可扩展性付出一点点代价（就是前面提到的那些限制）的话，请往下读。

帮助程序以一个Django项目框架（skeletal Django project）的形式发布，它包含了一个叫appengine_django的应用。你可以在这个骨架上编写新项目，或是把这个appengine_django应用文件夹复制到现有的项目里。

App Engine提供了一个小巧的管理员后台，叫做Development Console。安装帮助程序后，就可以通过URL:http://localhost:8000/_ah/admin（假设你的App Engine运行在8000端口上）来访问它。和普通的admin应用不同，这个工具只能用于数据库里已经有记录的模型（model）。如果你的模型还没有保存任何数据的话，就不能通过Development Console来创建新数据。

E.5 集成App Engine

在这一节里，我们要把第2章里开发的那个简单的blog移植到App Engine上去。我们打算遵循Google App Engine Helper for Django的README文件里列出的步骤来进行。这个文件可以在这里找到：http://code.google.com/p/google-app- engine-django/source/browse/trunk/README。在下面的例子中，我们假设项目的名字是mysite，应用程序的名字是blog。

把App Engine代码复制到项目里来

解压下载的帮助程序包后，你会得到一个名为 appengine_helper_for_django的目录。把目

录的内容复制到Django blog项目的主目录下，即app.yaml，main.py以及appengine_django目录同你的Django文件urls.py和 settings.py都在一个目录下。

现在编辑app.yaml文件，把应用程序名改为你在App Engine的Admin Console下注册的应用程序名。下一步就是要允许你的项目访问App Engine自身的代码。

> **注意**
>
> 这里把app.yaml改成App Engine里注册的名字实际上不是必须的，只有在你决定要把代码上传到App Engine的时候才需要。当代码运行在SDK的开发服务器上时，这个信息是不用填的。

如果安装App Engine SDK时用的不是Windows或是Mac OS X的安装程序，那你就得手动链接App Engine代码。在POSIX系统上（Linux或Mac OS X），可以调用ln命令：

```
$ ln -s THE_PATH_TO/google_appengine ./.google_appengine
```

集成App Engine Helper

下一步是要把帮助程序集成到负责指挥控制的manage.py里去。这样就可以和管理独立的Django应用一样管理我们的应用程序。方法是在文件开头的地方加上两行代码（指定shell的注释之后，导入execute_manager的代码之前），所以文件的前四行应该如下所示：

```
#!/usr/bin/env python

from appengine_django import InstallAppengineHelperForDjango
InstallAppengineHelperForDjango()

from django.core.management import execute_manager
```

这样App Engine就部分控制了一些Django的管理命令，并且可以加入App Engine相关代码来访问帮助程序的插件。不过，就算是修改过的manage.py，也依然能执行很多重要的命令。

有了这个“新的”manage.py后，我们就可以用它来生成一个新的settings.py文件。建议先备份原始的项目，在必要时还可以退回去！为什么呢？因为我们不得不去掉所有和App Engine不兼容的Django功能——我们之前描述的所有功能。方法是执行diffsettings命令。

```
$ manage.py diffsettings
WARNING:root:appengine_django module is not listed as an application!
INFO:root:Added 'appengine_django' as an application
WARNING:root:DATABASE_ENGINE is not configured as 'appengine'. Value overridden!
WARNING:root:DATABASE_%s should be blank. Value overridden!
WARNING:root:Middleware module 'django.middleware.doc.XViewMiddleware' is not
compatible. Removed!
WARNING:root:Application module 'django.contrib.contenttypes' is not compatible.
Removed!
WARNING:root:Application module 'django.contrib.sites' is not compatible. Removed!
DATABASE_ENGINE = 'appengine'
DEBUG = True
```

```
INSTALLED_APPS = ('django.contrib.auth', 'django.contrib.admin', 'mysite.blog',
'appengine_django')
MIDDLEWARE_CLASSES = ('django.middleware.common.CommonMiddleware',
'django.contrib.auth.middleware.AuthenticationMiddleware')
ROOT_URLCONF = 'mysite.urls'  ###
SECRET_KEY = 'w**sb^(p($wzxra*a9_@4_z0s(9i(9x3(w-aribbaaa4(r^wi'
SERIALIZATION_MODULES = {'xml': 'appengine_django.serializer.xml'}  ###
SETTINGS_MODULE = 'settings'  ###
SITE_ID = 1  ###
TEMPLATE_DEBUG = True
TIME_ZONE = 'America/Los_Angeles'
```

在输出中所有“WARNING”打头的行经常是指那些被移除的不兼容的内容或是不正确的设置。所有跟在WARNING和INFO之后的行则组成了新settings.py文件的内容。

将应用程序移植到App Engine里去

现在我们要调整一下应用程序来使用App Engine对象，首先是models.py文件，我们需要用appengine_django.models.BaseModel来替换 django.db.models.Model。备份你的模型文件，然后修改models.py如下：

```
from appengine_django.models import BaseModel
from google.appengine.ext import db

class BlogPost(BaseModel):
    title = db.StringProperty()
    body = db.StringProperty()
    timestamp = db.DateTimeProperty()
```

这里的代码是绝对相似的——这些对象和之前用到的对象基本是等价的。一个显著的区别是我们不再需要为title设置max_length。App Engine的StringProperty最大支持500字节。简单起见，这里body也用一样的属性（property）。

如果blog正文超出这个长度的话，你可以改用TextProperty。TextProperty可以储存超过500字节的内容，缺点是它不能被索引，也不能过滤和排序。好在通常你也不会去过滤或是排序body属性，所以问题也不是很大。

由于App Engine没有使用Django ORM，所以你必须把视图里的代码改成BigTable查询。App Engine数据库API提供了两种不同的查询方式：标准Query或是类似SQL的GqlQuery。因为我们偏爱的是Django ORM并且希望尽量保留高层次的数据访问，所以这里采用的是Query。

测试一下

现在是时候来测试一下这个翻新后的应用程序了。当运行manage.py runserver启动开发服务器时，你不会再看到Django的验证模型，它也不会再告诉你正在使用的是哪个设置文件或是展示友好的启动信息。取而代之的是App Engine服务器的输出。

```
$ manage.py runserver
INFO:root:Server: appengine.google.com
```

```
INFO:root:Running application mysite on port 8000: http://localhost:8000
```

打开浏览器窗口输入 http://localhost:8000/blog。你应该可以看到熟悉的blog画面，如图E.1。这里当然还没有显示任何帖子，因为你用的是App Engine的存储而不是原来Django ORM的数据库，所以当然是空的了。



图E.1 还没有文章的“新”blog

添加数据

之前说到过，App Engine里不能使用Django的admin应用。所以我们要自己手动添加第一篇blog帖子（通过Python shell），让App Engine的数据输入机制意识到这个特定模型（model）的存在。在这台机器上我们有安装IPython，所以可以看到它的启动信息和提示符。

```
$ manage.py shell
Python 2.5.1 (r251:54863, Mar  7 2008, 04:10:12)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.2 — An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.
```

导入BlogPost类，创建一个实例并填入详细信息。

```
In [1]: from blog.models import BlogPost
In [2]: entry = BlogPost()
In [3]: entry.title = '1st blog entry'
In [4]: entry.body = 'this is my 1st blog post EVAR!!'
In [5]: from datetime import datetime
In [6]: entry.timestamp = datetime.today()
```

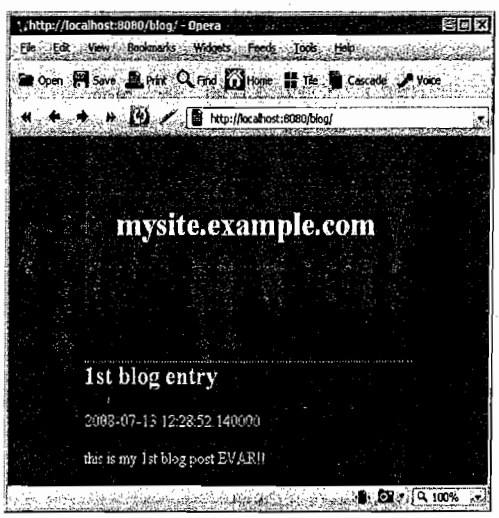
App Engine的DateTimeProperty能识别datetime.datetime对象，所以我们可以直接导入使用

它。然后调用today函数来获取当前的日期和时间并赋值给timestamp，接着就只需调用put方法保存对象就行了。

```
In [7]: entry.put()
Out[7]: datastore_types.Key.from_path('BlogPost', 1, _app=u'mysite')
In [8]: query = BlogPost.all()
In [9]: for post in query:
   ....:  print post.title, ':', post.body, '(%s)' % post.timestamp
   ....:
1st blog entry : this is my 1st blog post EVAR!! (2008-07-13 12:28:52.140000)
```

数据库API文档里说到，“Model类的all()方法返回了一个Query对象，它表示了一个取得所有对应类型的实体的查询”。（这段话引自数据操作文档，你可以在这里找到它：http://code.google.com/appengine/docs/datastore /creatinggettinganddeletingdata.html）就像你看到的一样，新添加的数据可以从shell里直接访问。剩下的就是将它在应用程序里显示出来了。

确认你的服务器还在运行中，刷新页面 http://localhost:8000/blog。你现在应该可以看到那篇帖子了，如图E.2所示。



图E.2 我们的blog有新文章了！

希望这段小小的例子能让你体会到Django admin的威力。当然了，同时你也看到了无论网站的状态如何，Python shell在操作数据时都是十分强大的。

E.6 为App Engine创建一个全新的Django应用

把现有的Django应用移植到App Engine实际上是一件相对麻烦的工作。比较起来其实大多数时候新建一个应用会简单一点——因为你不用“负担”一个现有的Django应用。如果你有了上一节的经验，下面这些在App Engine上创建全新Django应用的步骤是很容易的：

1. 和往常一样用django-admin.py创建Django项目。

2. 把App Engine代码（app.yaml，main.py，appengine_django）复制到项目目录里。

3. 用新的应用程序名修改app.yaml文件（就是向App Engine Admin Console注册的名字）。

4. 按需要链接Google App Engine的代码。

5. 运行 manage.py startapp NEW_APP_NAME。

6. 构建你的应用程序！

在构建应用的时候，记住不要编写“纯Django”的代码，比如要用App Engine Helper的模型（model）而不是Django的。不过有失也有得，你现在可以访问App Engine所有强大的API了。

- Python运行时（Runtime）
- 数据存储API
- 图形API
- 邮件API
- 内存缓存API Memcache API
- URL获取API（URL Fetch API）
- 用户API（Users API）

更多信息请参考App Engine的文档。

E.7 总结

这一节附录里，我们向你介绍了Google App Engine及其能力，和另一种开发Django应用的环境。虽然为了一些新功能，Django不得不牺牲一些自身的特性，这给Django程序员直接转向App Engine开发时增加了些许挑战。好在App Engine Helper的出现简化了这个任务。

随后我们深入展示了如何把一个应用示例移植到App Engine上去。以及通过一些简单的步骤创建一个全新的应用，而无须关心之前的代码。这里我们把如何将blog应用重写为一个100%的App Engine应用留给你作为练习。

最重要的是你作为一名Django程序员可以在继续编写Django的同时，享受到Google App Engine带来的好处：简单的部署，出色的可扩展性，以及利用Google成熟的基础设施的能力。

E.8 在线资源

下面是一些很有用的在线资源，完整的列表可以在本书的网站 withdjango.com 上找到。

- Google App Engine

http://code.google.com/appengine/

- App Engine SDK Project

http://code.google.com/p/googleappengine/

- App Engine Tutorial

http://code.google.com/appengine/docs/gettingstarted/

- Google App Engine Helper for Django

http://code.google.com/p/google-app-engine-django/

• Using the Google App Engine Helper for Django (Matt Brown, May 2008)

http://code.google.com/appengine/articles/appengine_helper_for_django.html

VIDEOS

• Rapid Development with Python, Django, and Google App Engine (Guido van Rossum, May 2008)

http://sites.google.com/site/io/rapid-development-with-python-django-and-google-app-engine

• Introducing GAE at Google Campfire (various, Apr 2008, 7 videos)

http://innovationstartups.wordpress.com/2008/04/10/google-app-engine-youtubes/

附录F 参与Django项目

Django不只是一个Web框架，它有的不仅仅是出色的设计和50000行代码。它背后有一整个社区的程序员，测试，翻译，回答问题的人和来自全球的志愿者。Django的AUTHORS文件列出了超过200名贡献者，此外还有很多为项目的进行作出或多或少贡献的人。

Django是开源项目的典范（Python之父Guido van Rossum语），它提供了很多方式让感兴趣的人加入进来。和其他很多人一样，作为一个Web程序员你会发现Django让你的工作变得更加简单而有趣，而这些随之而来的闲暇和快乐又能激发你回馈社区的灵感。下面是一些你可以贡献的方面。

最简单的方法甚至连编程都不需要：

- 提交改正文档里的笔误
- 在IRC频道和django用户邮件列表里回答新手的问题
- 帮助整理code.djangoproject.com上的bug报告，关闭无效的报告，并且帮助确认（或检验）有效的报告

如果你不介意动一下手的话，还有一些不需要太大工作量的选择：

- 为一个已知的bug提交patch
- 在不太流行的系统或是不常见（但是应该支持）的软件配置下运行Django测试来帮助界定潜在的问题
- 参与Django上某个正在开发新特性的分支

最后是一些比较艰巨的任务，它们会对整个Django社区产生深远的影响：

- 为Django中某个还未实现的语言提供本地化支持（如果你能找到的话！）
- 找到一个被广泛要求但是还未实现的特性，然后实现它
- 在Django上创建一个高质量的应用并且开源发布

如果你自己不确定能做什么，最好的方法就是先加入django-users或是django-developers用户列表或是在 irc.freenode.net上的#django IRC频道里花点时间到处看看。很快你就会开始为社区作出贡献了。如果你碰到了我们中的任何一个，别忘了打招呼哦！

> **注意**
>
> Django文档（在官网和withdjango.com上都有链接）有更多关于这方面的细节，从编码风格到邮件列表礼仪等等。

除了加入邮件列表和IRC频道之外，另一个成为Django社区成员的方法就是在http://djangopeople.net注册，这是一个全球性的Django程序员积聚地（如果你有兴趣组织一支

团队的话也可以通过这里来寻人）。另一个是Django程序员找工作的网站 http://djangogigs.com，它也有一个程序员的列表（http://djangogigs.com/developers.）。

最后，你还可以订阅Django官方博客，从那里可以获得很多关于Django开发的新闻。有关订阅的信息，以及更多其他社区工具的链接，都可以在以下网址找到：http://djangoproject.com/community/。

后　记

在编写这本书的时候，我们希望把它当作软件开发那样来对待。我们希望使用高质量的开源工具来确保工作顺利完成，特别是那些适合团队合作的工具。虽然这里没有什么激进的或是理想化的东西，不过在出版界的流程里居然还存在大量依赖某些私有的文字处理软件格式和email附件这种现象，我们实在希望能打破这种局面。以下是一份我们在准备、编写和编辑本书过程中用到的一些重要的开源工具。

用于控制我们的手稿和项目文件版本的软件是Subversion（也有一点Git和Mercurial）。虽然很难完整地说清楚到底版本控制对这类工作好在哪里，这里还是列出了一些：完整的项目历史；允许并行工作而不会影响到他人，甚至是同一份文件里的不同部分；任何时候都至少有四份完整的项目拷贝。

Trac是一个Python编写的轻型软件项目管理系统，它让我们可以方便地跟踪任何修改，另外还提供了基于wiki的共享笔记。Trac和 Subversion（或是其他任何通过插件支持的版本控制后台）非常强大，甚至对我们这样一个（很大程度上）非编程的项目来说也不例外。彩色的diff 能让你清楚地看到那天完成的工作量，以及在最后一轮修改中到底改变了哪些地方。

我们的手稿大多是多个由Win32或是Linux上的 Vim以及Mac上的TextMate创建的纯文本文件组成。它们由Markdown文本标记系统写成，可以方便地生成转换为HTML，PDF，和其他输出格式。选择Markdown是因为它良好的可读性和极少量的标记，这两点对写作都非常重要。我们用很简单的make工具，Markdown-Python（加上Wrapped Tables [wtables]扩展）来把文本文件（.txt或.mkd）编译成HTML。

我们用Mailman（Python编写的邮件列表管理器）设置的邮件列表来进行内部交流。Mailman除了能提供基本的邮件服务外，还能保存每条消息，这样我们就可以随时回去查看而不用担心要把它们保存到本地的email客户端上。

在编写一个复杂的软件系统时，操作系统一定要有一个成熟的软件包管理系统。能够安装SQLite、Memcached、Mako、PostgreSQL以及Apache等。从内建在Debian，Ubuntu和FreeBSD系统上出色的软件包管理系统，以及OS X上的MacPorts系统，我们获益良多。

Python语言在这里的重要性更是不言而喻。除了显而易见的优点，Python也支持了我们在质量控制上作出的努力。我们用简单的Python脚本解析手稿文件，通过doctest模块自动测试嵌在手稿里的交互Python示例，并且更新我们从更大的正常工作的应用程序里抓出来的代码示例。我们还用Python代码来扫描所有的手稿文件并更新对应的目录文件，这样就能在任何时候看到我们的进度。Python还被用来执行Markdown-Python编译器从而将所有的文本文件转换成HTML。我们对能在这项工作中如此大量的使用Python都感到非常高兴，希望你也一样。

我们的Makefile还包含了一些整理生成内容的指导，例如把所有的HTML合并为一个文件，压缩生成ZIP文件，为每个HTML文件打开新的浏览器窗口，以及生成PDF文件等（通过html2ps［不是PHP的那个，是另一个］和Ghostscript的ps2pdf的帮助）。

最后，本书的网站也是用Django开发的哦！

软 件	链 接
Subversion	http://subversion.tigris.org
Trac	http://trac.edgewall.org
Mailman	http://www.gnu.org/software/mailman
Markdown	http://daringfireball.net/projects/markdown
Markdown-Python	http://freewisdom.org/projects/python-markdown
wtables	http://brian-jaress.livejournal.com/5978.html
make	http://www.gnu.org/software/make/
TextMate	http://macromates.com
Vim	http://www.vim.org
Ghostscript	http://pages.cs.wisc.edu/~ghost/
html2ps	http://user.it.uu.se/~jan/html2ps.html
Firefox	http://mozilla.com/firefox
Ubuntu	http://ubuntu.com
FreeBSD	http://freebsd.org
Macports	http://macports.org
Python	http://python.org
Django	http://djangoproject.com

封面上的大桥是位于荷兰鹿特丹的Erasmus大桥。

荷兰也是Python之父Guido van Rossum的出生地。而Django，也希望能作为一座桥梁，把广阔的Web应用开发世界和那些希望在网上发表作品的人联系起来，让他们不用去关心如何编写复杂的服务器代码，SQL语句，或是“MVC”是什么意思等。